September 1990

UILU-ENG-90-2242
DAC-25

*Analog and Digital Circuits*





# iCHARM: HIERARCHICAL CMOS CIRCUIT EXTRACTION WITH POWER BUS EXTRACTION

**Russell Makoto Iimura**



*Coordinated Science Laboratory*
*College of Engineering*
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS<br>None |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for public release; distribution unlimited |
| 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>UILU-ENG-90-2242 (DAC-25) | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION<br>Coordinated Science Lab<br>University of Illinois<br>6b. OFFICE SYMBOL (If applicable)<br>N/A | 7a. NAME OF MONITORING ORGANIZATION<br>Office of Naval Research &<br>Rome Air Development Center |
| 6c. ADDRESS (City, State, and ZIP Code)<br>1101 W. Springfield Ave.<br>Urbana, IL 61801 | 7b. ADDRESS (City, State, and ZIP Code)<br>Arlington, VA 22217<br>Griffiss Air Force Base, NY 13441-5700 |
| 8a. NAME OF FUNDING / SPONSORING ORGANIZATION Joint Services Electronics Program & RADC<br>8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>F30602-88-D-0028 N00014-90-J-1270 |
| 8c. ADDRESS (City, State, and ZIP Code)<br>Arlington, VA 22217<br>Griffiss Air Force Base, NY 13441-5700 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)

iCHARM: Hierarchical CMOS Circuit Extraction with Power Bus Extraction

12. PERSONAL AUTHOR(S)

Iimura, Russell Makoto

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| Technical | FROM 1988 TO 1990 | 90 Sep 5 | 110 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Circuit extraction, hierarchical extraction, circuit parasitics, reliability estimation, database, design framework. |
| | | | |
| | | | |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Circuit extraction is critical in the validation of VLSI circuits since it provides the link between the design and the simulation phases. The use of hierarchical design techniques and hierarchical analysis methods increases design productivity. In this thesis, the development of iCHARM, a hierarchical circuit extractor, is described. The extractor takes its input layout in either the CIF format or the Oct VLSI database format. The extractor produces circuit parasitics including capacitances and resistances. A power bus extraction mode has been developed to calculate power bus currents for reliability estimation. The primary contribution of this work is a method to extract a circuit hierarchically without flattening and with minimal overhead. A full-chip layout was used to test the extractor's functionality and to allow a comparison of the hierarchical and flat extraction modes.

| | |
|---|---|
| 20. DISTRIBUTION / AVAILABILITY OF ABSTRACT<br>☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL |

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

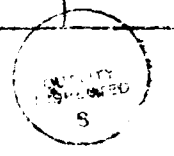
# iCHARM: HIERARCHICAL CMOS CIRCUIT EXTRACTION WITH POWER BUS EXTRACTION

BY

RUSSELL MAKOTO IIMURA

B.S., University of Colorado, 1984





THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

# ABSTRACT

Circuit extraction is critical in the validation of VLSI circuits since it provides the link between the design and the simulation phases. The use of hierarchical design techniques and hierarchical analysis methods increases design productivity. In this thesis, the development of iCHARM, a hierarchical circuit extractor, is described. The extractor takes its input layout in either the CIF format or the Oct VLSI database format. The extractor produces circuit parasitics including capacitances and resistances. A power bus extraction mode has been developed to calculate power bus currents for reliability estimation. The primary contribution of this work is a method to extract a circuit hierarchically without flattening and with minimal overhead. A full-chip layout was used to test the extractor's functionality and to allow a comparison of the hierarchical and flat extraction modes.

# ACKNOWLEDGEMENTS

I would like thank my advisor, Professor Ibrahim Hajj, for his patience and guidance throughout the development of my graduate work. This work owes a great deal to Krishna Belkhale who developed the basis for the extractor program described here. I also would like to recognize my colleagues and officemates in the Circuits and Systems Group for their assistance and helpfulness. Finally I would like to thank my family for their encouragement and example, and the friends that I have made at the University of Illinois, especially Holly Sakane, for all of their support through the stresses of the final months.

This work was funded by the Joint Services Electronics Program and by the Rome Air Development Center of the U.S. Air Force.

# TABLE OF CONTENTS

# CHAPTER 1

# INTRODUCTION

## 1.1. Design of VLSI Circuits

The high level of integration in recent VLSI circuits — up to one million transistors in a single chip [7] — has required the use of increasingly sophisticated Computer-Aided Design (CAD) tools. These CAD programs are used to analyze and verify the circuit and layout design of a chip before it is actually made, since any changes after fabrication add significantly to the development cost of the chip. In addition to ensuring that a chip works "in first silicon," the use of automated methods also shortens the chip's "time-to-market" which can add to its profitability.

This thesis is concerned with the problem of circuit extraction, which is the transformation of layout information into circuit information. After extraction, the circuit is simulated before it is fabricated to verify that it performs as intended.

In addition to CAD, the use of hierarchy is a way to increase design productivity and reduce the complexity of the design task. Rather than design each individual transistor, the repetition or regularity of a design is exploited to reduce the amount of work that must be done. A *cell* is defined to be the unique or atomic part of a design that needs to be designed only once. Copies or *instances* of cells are made wherever the cell is needed. There are certain so-called *primitives* of a cell: in circuit layout, the primitives are the mask-level geometries. Cells may contain primitives, or instances of other cells, or a combination of both. Cells that contain no instances are called *leaf-level* cells; cells that contain instances but are not the top-level cell are often called *intermediate-level cells*. A hierarchically defined cell is shown in Figure 1.1. In the figure, cell A is the top-level cell, cells B, C, and D are intermediate cells, and cells E and F are leaf-level cells. The use of hierarchy in a top-down design style is particularly effective.

A top-down, hierarchical design style, as shown in Figure 1.2, involves decomposing high-level functional blocks into smaller and smaller functional units until each unit is well-defined, is a reasonable size for a leaf-level block, and has an obvious implementation in the target

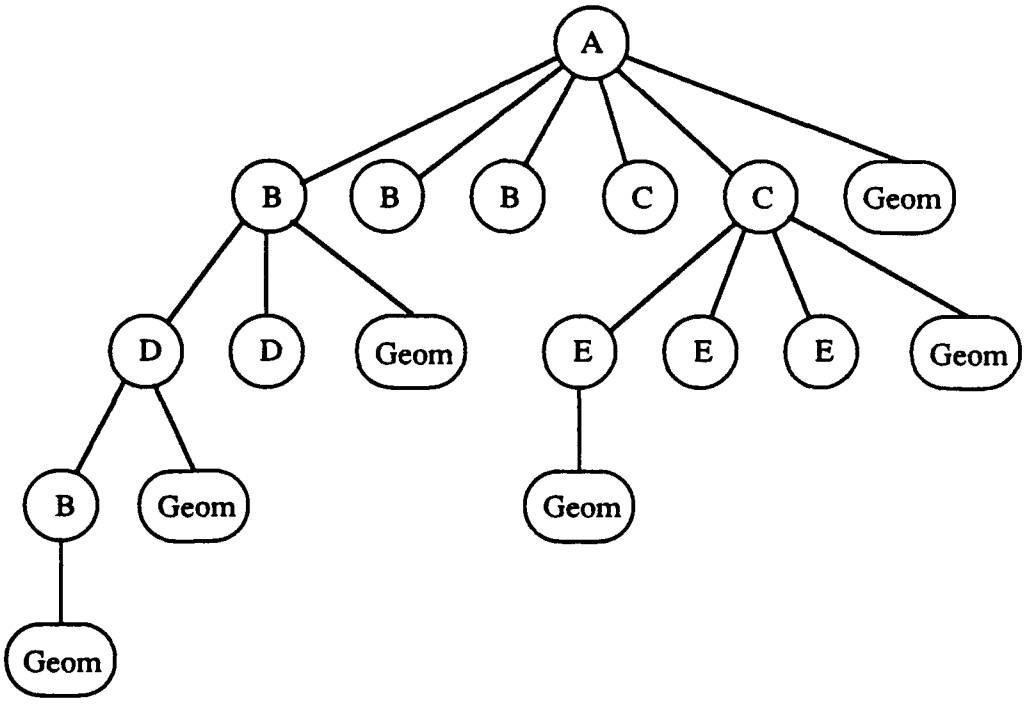



**Figure 1.1 - A hierarchically defined cell**

technology (for instance in MOS, bipolar, or both). At each design level the current implementation is verified by creating models of the design and simulating them. At the architectural level, the models are behavioral; at the logic level, logic gates are simulated; the design is finally composed of transistors at the circuit design level. The circuit's actual implementation on silicon is not realized until the layout or artwork design work is completed.

Circuit extraction provides the link between the design of artwork and its verification through simulation. The circuit elements extracted from the actual artwork must match the behavior that was previously simulated at the other levels in the top-down design process. Only when this is achieved is it felt that there is sufficient reason to believe the chip will operate correctly when fabricated.

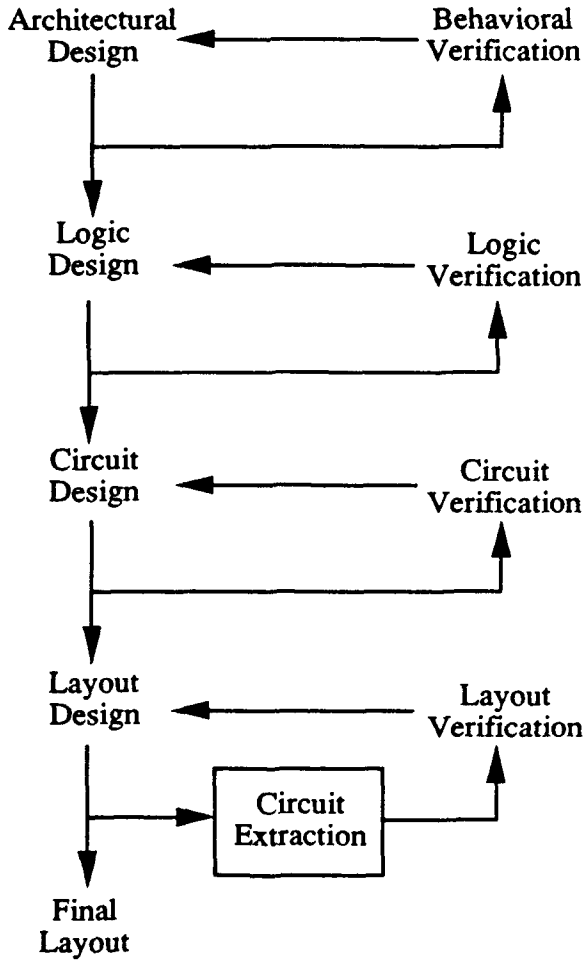



**Figure 1.2 - A top-down, hierarchical design process**

## 1.2. Circuit Extraction

The artwork provides a large measure of reality in verification; at other levels the difference between the models and reality may be significant if the modeling is not done properly. Therefore, it is important to accurately model the circuit implemented by the artwork. There are two main goals for verification:

(1) functional verification

(2) performance verification

Functional verification ensures that the circuit at the current design level accomplishes its desired function. Performance verification goes one step further to see if the circuit operates within its timing requirements. Estimating the delays of the circuit is dependent on the design style used as well as the technology and process of the circuit. For example, a circuit may be implemented in a design style using fully complementary CMOS or Domino CMOS to trade off ease-of-design against speed.

To verify the circuit, extraction achieves the following:

(1) identification of active devices (transistors)

(2) identification of electrically connected geometries (nets)

(3) extraction of the electrical parameters of transistors (e.g., dimensions of the transistors)

(4) estimation of parasitic resistances and capacitances of the interconnection nets

The identification of transistors and nets is required to functionally verify the circuit. The structural connectivity is the minimum required to verify a circuit's function.

On the other hand, the extraction of transistor parameters and the parasitic resistance and capacitance of the nets connecting the transistors is needed for accurate performance verification. The resistances and capacitances of the interconnection lines are termed *parasitic* since they are not explicitly designed as part of the circuit but are a side effect of the electrical properties of the conductor materials. As the dimensions of typical artwork shrink below one micron, the parasitic effects begin to be more significant on circuit performance. The interconnection lines will then begin to occupy more of the chip area relative to the size of the active devices. As the minimum line widths shrink, the length and thus the resistances of the interconnection lines will increase. For these reasons, it is critical to accurately extract the parasitic effects from the circuit layout for performance verification.

## 1.3 Features of iCHARM

This thesis describes the development of a hierarchical circuit extractor named iCHARM.[1]

---

[1] Since the extractor is hierarchical, the natural name for it could be HEX, for Hierarchical EXtractor. HEX has already been used to describe an extractor in the past, so a synonym for "hex," "charm," was chosen.

Just as hierarchy is used to reduce the amount of work required to design a VLSI circuit, the work done by an extractor can make use of the regularity of the circuit to reduce the analysis time. In fact, the hierarchical partitioning of the circuit by the designer is also used to do the extraction. *Hierarchical* extractors analyze each unique cell of the circuit only once and refer to the previously extracted information when an instance of the cell is encountered. When the analysis is done *flat*, on the other hand, each part of the circuit is looked at with no attempt to recognize any similarity with previously analyzed structures.

The iCHARM program was developed using PACE, an existing extractor program written by Krishna Belkhale, as its basis [1]. PACE contributed the flat extraction of transistors and the parasitic extraction code to iCHARM, and was implemented on a parallel computer. The ability to do hierarchical analysis and the enhanced input routines to read a layout from the Oct database format are new contributions and are the goals of this research.

### 1.3.1. Capabilities of iCHARM

iCHARM is written in C, runs under UNIX[2] and has the following features:

**CMOS circuits:**

The extractor assumes the input layout is in CMOS technology with any number of metal interconnection layers. A technology file is read to determine the process parameters of each layer and the connectivity between layers.

**Manhattan geometries:**

iCHARM supports only rectangles as mask geometries; therefore, the extractor works only for *manhattan-style* layouts, i.e., layouts where the edges of the geometries are parallel to the x- or y- axis.

**CIF and Oct input formats:**

The input layout read by iCHARM is described in either the CIF (Caltech Intermediate Form) format or in the Oct database format. The two formats and their input routines are described in more detail in Chapter 2.

**Spice output format:**

The extracted circuit description is output in the Spice circuit simulator format. If the

---

[2] UNIX is a trademark of AT&T Corp.

extraction is done hierarchically, each cell extracted generates a .SUBCKT card. An option to output the circuit in the Oct format also is envisioned.

**Scanline algorithm:**

The input layout is represented internally in iCHARM in terms of rectangles. A scanline algorithm is employed to carry out the extraction process. This is described in detail in this thesis.

The program may be run in several modes as shown in Figure 1.3, and described as follows:

(1) First, the user has the option of running the extractor in flat or hierarchical extraction mode. If there is a high degree of regularity in the circuit (if the circuit has just a few cells but perhaps a large number of instances), then the hierarchical mode should run faster and use less run-time memory than an equivalent run in the flat extraction mode.

(2) The user also has a choice of the parasitics of the interconnection lines that may be extracted. The default mode is to extract each net's substrate capacitance (the capacitance

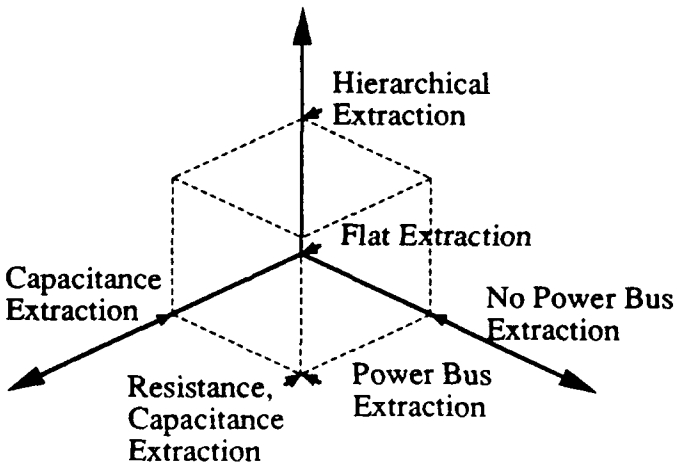



Figure 1.3 - Run-time options of iCHARM

between the net and ground). In addition to the capacitance, the user may extract the resistance through each interconnection net.

(1) iCHARM has a special mode to extract the mask-level rectangles of the power bus nets (typically Vdd and GND). This mode does additional processing of the power bus nets for subsequent analysis by other programs to do reliability estimation. This is described in Section 4.3.

## 1.4. Overview of the Thesis

This thesis consists of six chapters. Chapter 2 presents the input routines that read the input layout and create hierarchical data structures. iCHARM reads the layout description in either the CIF or Oct data formats. Chapter 3 describes the circuit extraction process in more detail. Various extraction algorithms are presented along with the method employed by iCHARM, a scanline algorithm. The implementational details of iCHARM are split into Chapters 4 and 5. The flat extraction techniques to extract the basic connectivity and circuit parasitics are presented first. Hierarchical extraction techniques are covered in the Chapter 5. In Chapter 6, the results of the testing of iCHARM using a complete chip layout are presented. The thesis ends with a list of possible extensions to the extractor and some concluding remarks.

# CHAPTER 2

# INPUT TO CIRCUIT EXTRACTION

The input to an extractor consists of the mask geometries of the layout. In the case of iCHARM, the layout consists solely of rectangles. iCHARM supports two input formats to describe the layout: the CIF format and the Oct database format. During the input phase of iCHARM, the input layout is read in and internal data structures for the program are created. In this chapter, the hierarchical data structures that are created from the layout are described first. Then, the two input formats are presented along with a brief description of the procedure used to create the data structures from each of the input formats.

## 2.1. Hierarchical Data Structures

In order to do hierarchical extraction, it is important for iCHARM to preserve the hierarchical structure of the layout that was created by the user. In iCHARM, the two data structures representing the hierarchical layout are the *cell* and *instance* structures. The C-language definitions of these basic structures are shown in Figure 2.1.

The cell struct contains the definition of a cell. The cell number (or symbol index) serves as the identifier for the cell. The *rectlist* field of the cell contains a list of the layout rectangles for the cell. The bounding box for the cell, the smallest rectangle that encloses all of the layout rectangles of the cell, is computed when all of the rectangles of the cell are read in. The coordinates of the cell bounding box are placed in the *bound* array. The *instlist* field of the cell has a list of the instances of the cell. The extraction process produces a list of nets and transistors for the cell (the *netlist* and *tranlist* fields); the other fields of the cell struct will be explained later as needed.

Pointers to all of the cell definitions are kept in a hash table. The hash function uses the *cellnum* field to identify each cell. In this way, any cell definition can be retrieved given a cell's *cellnum*.

```
/* C E L L - the definition of a hierarchical cell */
typedef struct _cell {
   int cellnum;                  /* identifying number */
   char *cellname;               /* name string */
   boolean done;                 /* if cell has been processed */
   long bound[4];                /* bounds of the cell */
   struct _rect *rectlist;       /* list of rects for the cell */
   struct _label *labellist;     /* list of labels for the cell */
   struct _rect *ovlprectlist;   /* list of rects overlapping this cell */
   struct _term *exttermlist;    /* list of external terminals */
   struct _term *inttermlist;    /* list of internal terminals */
   struct _inst *instlist;       /* list of instances in this cell */
   struct _net *netlist;         /* list of nets in this cell */
   struct _tran *tranlist;       /* list of transistors in this cell */
   struct _cell *nextptr;
} cell, *cellptr;

/* I N S T A N C E - the hierarchical instantiation of a cell */
typedef struct _inst {
   struct _cell *defn;           /* pointer to the cell definition */
   int tfmatrix[3][3];           /* transform matrix: no non-90° rotations */
   long bound[4];                /* bounds of the instance */
   struct _term *termlist;       /* list of terminals for this instance */
   struct _inst *nextptr;
} inst, *instptr;
```

**Figure 2.1 - Cell and instance data structures**

An instance struct is created for each instantiation of a cell. The purpose of the instance struct is to prevent the repetition of information contained in the cell definition, thus a pointer back to the cell definition is needed (this is the *defn* field). The transformation matrix of the instance — a specification of how the instance is placed within its parent — is in the *tfmatrix* field of the instance. Transformation matrices are presented in Section 2.3. The cell's bounds are transformed by the transform matrix into the *bound* array of the instance; thus the bounding box of the instance is always available. In Figure 2.2, a given hierarchy and its cell and instance structures are shown as an example.

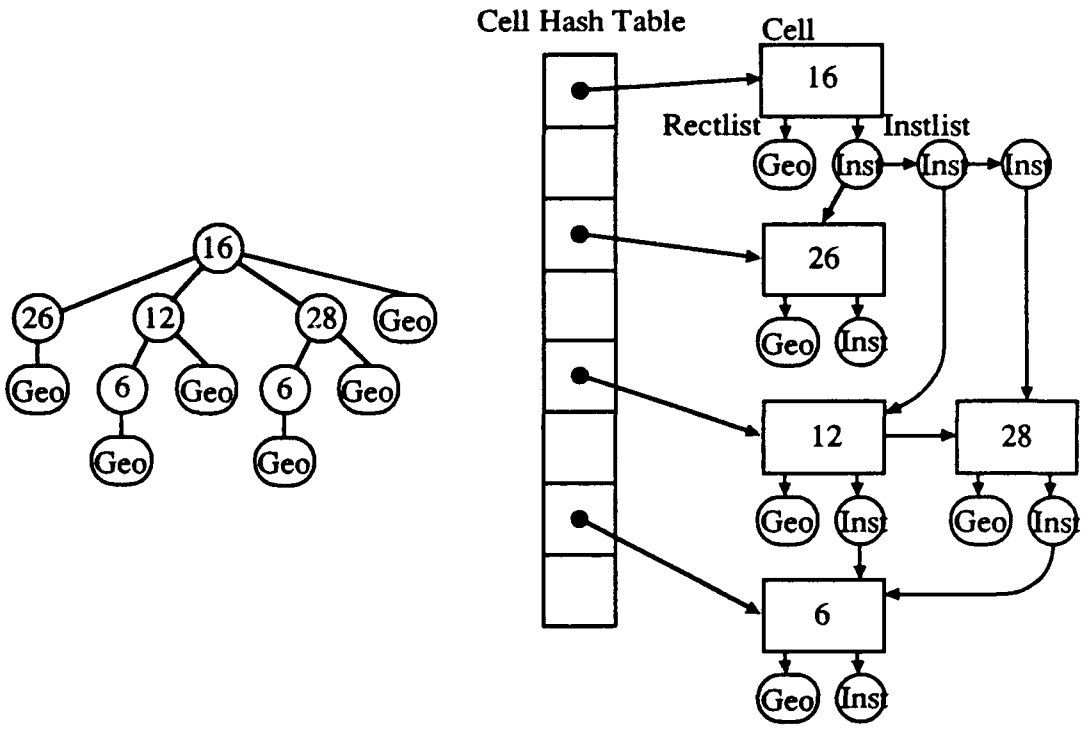



**Figure 2.2 - Hierarchical data structure example**

## 2.2. CIF Input Format

For many extractors including iCHARM, the input layout is described in CIF (Caltech Intermediate Form — Version 2.0), an ASCII file format where each layout geometry is described in terms of its shape (in iCHARM this is restricted to rectangles), mask layer, and coordinates. A detailed description of the CIF format is given in Mead [9]. In the following section, the basic features of the CIF format will be presented in order to discuss how the hierarchical data structures are built during the input phase of iCHARM.

A CIF file describing a layout is composed of a sequence of commands, the last of which is an end marker. Each command is terminated with a semicolon. The CIF commands recognized

| Table 2.1 - CIF Commands Recognized by iCHARM[1] | |
|---|---|
| Command | Syntax |
| Box with length, width, center | B length width point point; |
| Layer specification | L layername; |
| Start symbol definition with index, scale a/b | DS index a b; |
| Finish symbol definition | DF; |
| Call symbol | C index transformation; |
| User extension | digit user_text; |
| Comments | (comment_text); |
| End marker | E |

by iCHARM — a subset of all of the commands described in Mead [9] — are shown in Table 2.1.

The rectangle primitives are defined by the Box command. The Layer command defines the current mask layer; all primitives that follow the Layer command are to be placed on the current mask layer.

CIF allows cells (or *symbols* in CIF terminology) to be defined to create hierarchy in a design and reduce the length of the file. The DS command begins the definition of a cell, and the DF command concludes that definition. All primitives between these two commands belong to the cell. The cell number (or index) that identifies the cell is the first parameter of the DS command.

The C command is used to make an instance (or *call*, in CIF terminology) of the given cell (identified by the symbol number) and to apply the given transformation to the primitives within the cell definition. There are four types of transformations that may be applied to coordinates within the cell when it is instantiated; these are shown in Table 2.2.

The primitives in the instantiated cell are transformed in the *order* of the transformations given in the cell call command. For example, "C 23 T 500 0 MX;" adds 500 to the x-coordinates

| Table 2.2 - Primitive Transformations | |
|---|---|
| T point | Translate the called cell's origin to this point |
| M X | Mirror in X; i.e., multiply the x-coordinate by -1 |
| M Y | Mirror in Y; i.e., multiply the y-coordinate by -1 |
| R point | Rotate the cell's x-axis to this direction vector |

[1] - After Mead [9], page 116.

of the primitives in cell 23, then multiplies the x-coordinates by -1; however, "C 23 MX T 500 0;" does the multiplication by -1 before the translation of 500.

When a cell is instantiated, each transformation is not performed separately but can be accomplished in one operation through the use of a single *transformation matrix*. A 3 × 3 transformation matrix T is used to transform a point (x, y) in the cell definition to its instantiated coordinates (x′, y′) in the final design by the matrix operation

$$[x' \; y' \; 1] = [x \; y \; 1]T$$

The transform matrix T should be the product of primitive transformations given in the cell call in the same order given in the call. For example, if $T = T_1 \, T_2 \, T_3$, then $T_1$ is the primitive transformation matrix for the first transformation in the cell call, $T_2$ is the matrix for the second transformation, and $T_3$ is the matrix for the third. Each primitive transformation is found by the following template:

$$\text{T } a \; b \quad T_i = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{matrix} \qquad \text{M Y} \quad T_i = \begin{matrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$\text{M X} \quad T_i = \begin{matrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} \qquad \text{R } a \; b \quad T_i = \begin{matrix} a/c & b/c & 0 \\ -b/c & a/c & 0 \\ 0 & 0 & 1 \end{matrix}$$

where $c = \sqrt{a^2 + b^2}$.

## 2.3. Transformation of Nested Cells and Flattening the Hierarchy

A cell definition may contain calls to other cells, and these cells may in turn contain calls to other cells, etc. In order to transform coordinates in a nested subcell, it is necessary to combine the effects of the transformation matrices of the instances throughout the hierarchy. For example, suppose a chip has an instance of cell A, and cell A has an instance of cell B (Figure 2.3 illustrates the situation). The instance of A and the instance of B have transformation matrices $T_A$ and $T_B$, respectively. In order to transform the coordinates of a rectangle in cell A to the coordinate system of the chip, the transform matrix $T_A$ may be used; similarly, to transform coordinates in cell B to cell A requires $T_B$. To transform coordinates in cell B *two levels* to the coordinate system of the chip requires:

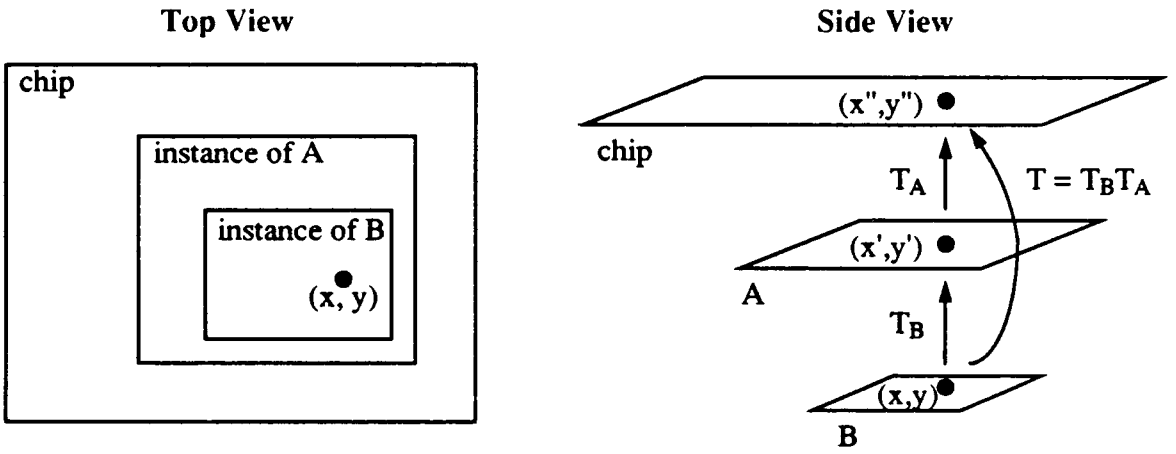



**Figure 2.3 - Transformations in a two-level hierarchy**

$$[x' \; y' \; 1] = [x \; y \; 1] \; T_B \; T_A$$

One can see that the transformation of two levels from the coordinate system of cell B to the coordinate system of the chip may be combined into a single transformation matrix of $T = T_B \; T_A$. Combining the effects of several levels of transformation matrices is useful in flattening the circuit hierarchy.

Flattening one level of hierarchy involves instantiating all rectangles of a cell instance into its parent cell. To instantiate a rectangle from an instance into its parent, first a copy is made of the rectangle. The instance's transform matrix is then used to transform the rectangle from the instance's coordinate system to the parent's coordinate system. To flatten through all levels of hierarchy, one starts at the leaf-level cells and instantiates their rectangles into their parent cells. This continues up through all levels of the hierarchy until the rectangle is instantiated into the top-level cell.

The flat extraction mode in iCHARM first flattens the hierarchy of the circuit so that all rectangles in the instances below the top-level cell are instantiated into the top-level cell. Extraction is then done on the rectangles present in the top-level cell, which now holds all of the

rectangles for the circuit. If there is a large degree of repetition in the top-level cell, a large amount of memory would be required to store the design. In any case the flattening procedure is simple to execute.

The procedure to flatten a given hierarchy, as outlined in Procedure 2.1, is a recursive walk down the hierarchy starting at the top-level cell. At each level, the current transformation matrix is used to transform the coordinates from the current level to the top-level coordinate system. To descend to the next level (for the next recursive call), the current transform matrix is multiplied

---

**Procedure 2.1 - Flattening a hierarchy**

```
{
  Flatten()
  Input: Top-level hierarchical cell: Cellhead.
  Output: All rects of lower-level cells instantiated into Cellhead.
}
Flatten(Cellhead)
begin
  for all instances, instp, of Cellhead
        InstantiateCell(instp→defn, instp→tfmatrix);
end;

{
  InstantiateCell()
  Input: currentcell: current cell being instantiated
         tmatrix: current transformation matrix
  Output: The rects of lower-level cell instances have been instantiated into Cellhead.
}
InstantiateCell(currentcell, tmatrix)
begin
{ copy all of currentcell's rects into Cellhead }
  for all rects, currrect, of currentcell begin
        transform coords of currrect using transform matrix tmatrix;
        put currrect into Cellhead's rectlist;
  end;
  for all instances, instp, of currentcell begin
  { premultiply current transform matrix by instp's transform matrix }
        newtmatrix ← MatrixMultiply(instp→tfmatrix, tmatrix);
        InstantiateCell(instp→defn, newtmatrix);
  end;
end;
```

---

with the transform matrix of the instance that will be used for the next level of recursion. In this manner, the effects of all the previous transform matrices (from the top-level down to the current level) are combined into the current transform matrix, so that it may be used to transform the current coordinate system into the top-level coordinate system.

## 2.4. CIF Input Module

Reading in a CIF layout is straightforward since, by the rules of the language, a cell must be defined before it is called. Cells are defined in bottom-up order in a hierarchical CIF file. Therefore, cell structs are created in the order they are defined in a CIF file: for each DS command, a new cell struct is allocated. The rectangles labels that follow the DS command are attached to the cell until a DF command is read.

Any cell instances that occur within the cell ("C" commands) create instance structs that are attached to the cell. The *defn* pointer of the instance struct is set by retrieving the cell definition of the instance, since the cell must be previously defined. Any CIF primitive transformations ("T", "M", or "R") in the cell call are converted into one transformation matrix which is attached to the instance struct.

## 2.5. Shortcomings of CIF

The purpose of the CIF format is to provide a standard interchange format between various programs to describe mask layout data as shown in Table 2.3. CIF may be produced as the output of a symbolic layout tool or an interactive layout editor which originally created the layout. The layout described in CIF may then be processed by a variety of programs: display or plotting programs to view or produce a hard-copy of the layout, pattern generators to produce the final

**Table 2.3 - Programs That Produce or Consume CIF**

| Input layout programs | → CIF → | Programs to process layout |
|---|---|---|
| layout editor | | layout plotter |
| symbolic layout tool | | layout display |
| | | pattern-generators (for final mask creation) |
| | | layout verification and analysis programs |

masks, and programs to verify or analyze the layout such as extractors and reliability analysis or yield prediction programs.

iCHARM needs to support CIF because it provides this standard interchange format and also because many extractor programs use CIF. Existing layouts described in CIF can be used to compare results against other extractors.

One shortcoming of CIF, however, is that it is not meant to be a design language. The design of a layout is not meant to be carried out by manipulating a CIF file directly. CIF is not an efficient way to internally represent layout for a program, i.e., CIF is not a good database or secondary storage format for a layout. CIF is meant to be produced by layout programs for subsequent use by other layout programs since the internal representations of the layout in the communicating programs are dissimilar. The main shortcoming of CIF as an input format — its loose coupling with the design of the layout — necessitates the use of the Oct/Vem system. This is described in the next section.

## 2.6. The Oct/Vem System

Oct/Vem is a VLSI data management and design system developed at UC Berkeley [12]. *Oct* is a data management system capable of storing the design data for an entire VLSI chip, including the schematics, artwork, or the extracted netlist of a design. The design information in the Oct format is stored in binary files, and Oct provides a simple interface for different application programs to access the design data.

*Vem* is a graphical editor used to manipulate the graphical representations of design data in the Oct format; since Oct can store circuit schematic or mask artwork information, these representations may be created and edited with Vem. Vem features a menu-based, multi-window user interface that is built using the X-window system.

Figure 2.4a shows a typical design verification system that uses CIF as an interchange format between the layout editor and extractor. Spice format files are used to communicate the extracted circuit to a simulator. This verification system uses different file formats to transfer data between each analysis program. This means that considerable effort is spent "re-inventing the wheel" while each program writes code that parses its input format and then writes out another output format. In addition, as the size of a design gets very large, keeping track of all design revisions becomes a difficult task. The situation is further complicated by the fact that multiple files for each revision must be maintained.

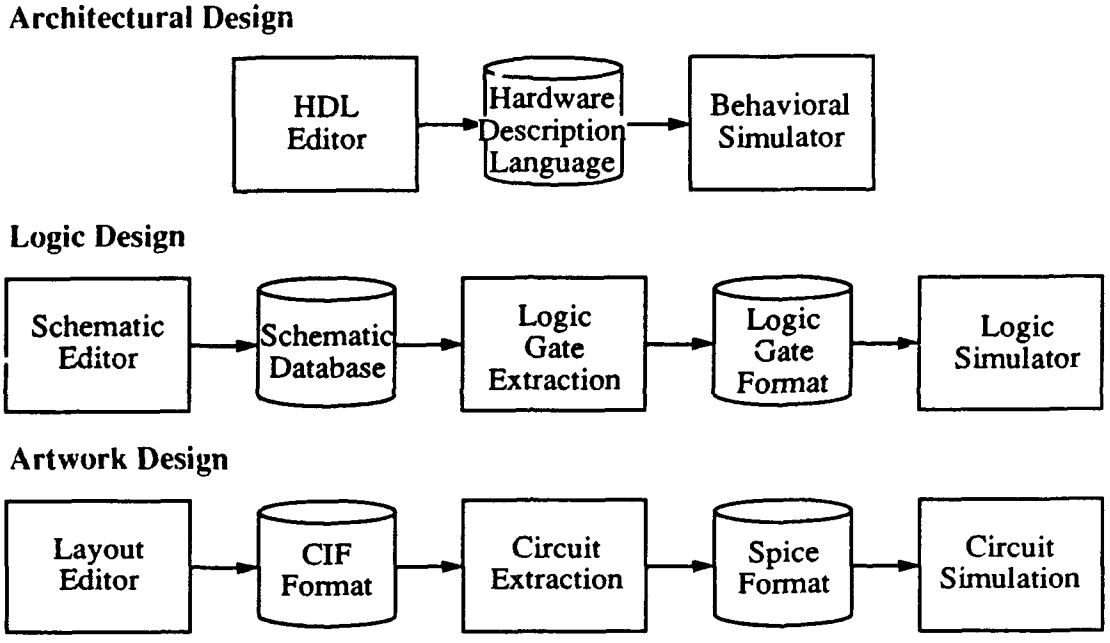



**Figure 2.4a - A design system using many interchange formats**

Figure 2.4b shows a unified, integrated environment for design with an Oct database as its central element. Since an Oct database can provide a repository for all design data — artwork as well as an extracted netlist for simulation — both artwork and simulation tools can access the design database through the same interface. The same access routines are used by each application program to access Oct design data. All data conversions between programs that require different file formats would be eliminated. In addition, since designers are spared the juggling of files in many different formats, data management problems are kept at a minimum.

For these reasons, iCHARM accepts input layouts in the Oct format as well as in the CIF format. In the next section, the Oct format is introduced with the goal of showing how an application program — in this case the extractor iCHARM — uses the Oct access routines to read layout information in the Oct format.

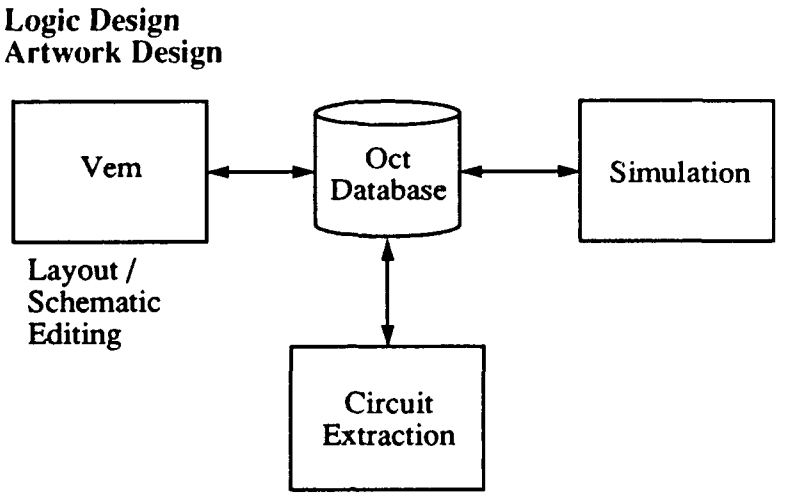



**Figure 2.4b - An integrated design system using Oct/Vem**

## 2.7. The Oct Database and Terminology

Design data in the Oct format are organized and specified by the concepts of a cell, view, and facet. The basic unit of storage in Oct is a *cell*, which, for example, may contain a single transistor, a logic gate, or an entire functional unit (e.g., ALU, CPU, or RAM). To efficiently represent a design, Oct cells are hierarchical so that one cell may contain instances of other cells.

Each cell in Oct may have one or more *views* depending on the point at which the cell is in the design process. For example, at the architectural or logic design level, the cell is best described by its schematic view, which abstractly defines the way instances of the cell are interconnected. On the other hand, a symbolic view would show the relative placement of objects within the cell. At the artwork design level, a physical view describes the mask layout for leaf-level cells. There also could be a simulator view of a cell that has objects of interest for simulation, namely, net and component lists. This view of the cell would obviously be created by an extractor.

Finally, for each cell and view there can be multiple representations or *facets*. The default facet is the *contents* facet. In a schematic view, for example, the nets and instances of a cell are

included in the contents facet. For the physical or layout level view, the contents facet would contain all of the mask geometries of the cell. Different facets for a view may be created to limit the amount of detail used to represent the design. Another facet may be created to show a simplified aspect of the same view: for a schematic view an *interface* facet may show only the bounding box of the cell along with the interconnection terminals at the edge of the cell. In Figure 2.5, a contents and interface facet for a schematic view is shown. This simplified abstraction of the view may be used to speed up the processing of a design hierarchy when a detailed representation of each cell is not needed.

The facet is the object that is edited in the Oct/Vem system. For instance, in Vem, the <cell>:<view>:<facet> must be specified to edit an object. The actual design data for a facet are stored in a binary-encoded file named with the name of the facet. Since each cell can have multiple views and each view can have multiple facets, the <cell>:<view>:<facet> is stored in the file named <facet> under the directory named <view> which in turn is under the <cell> directory. Figure 2.6 shows the cell "alu186" with the views "schematic," "physical," and "simulator." Each view may have both a contents and an interface facet.

Each facet contains a collection of primitives or *octObjects* that constitute the design. A facet itself is an octObject. The octObject primitives can represent structural information

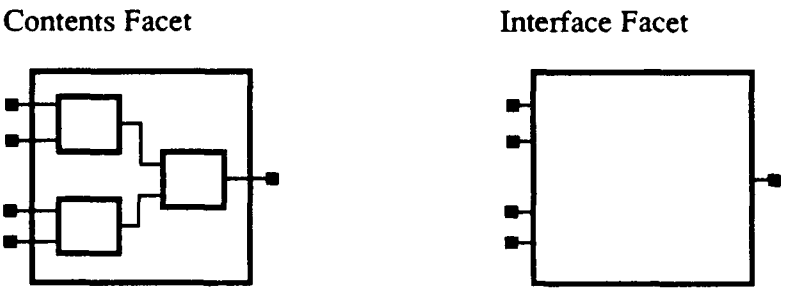



**Figure 2.5 - Contents and interface facet of a schematic view**

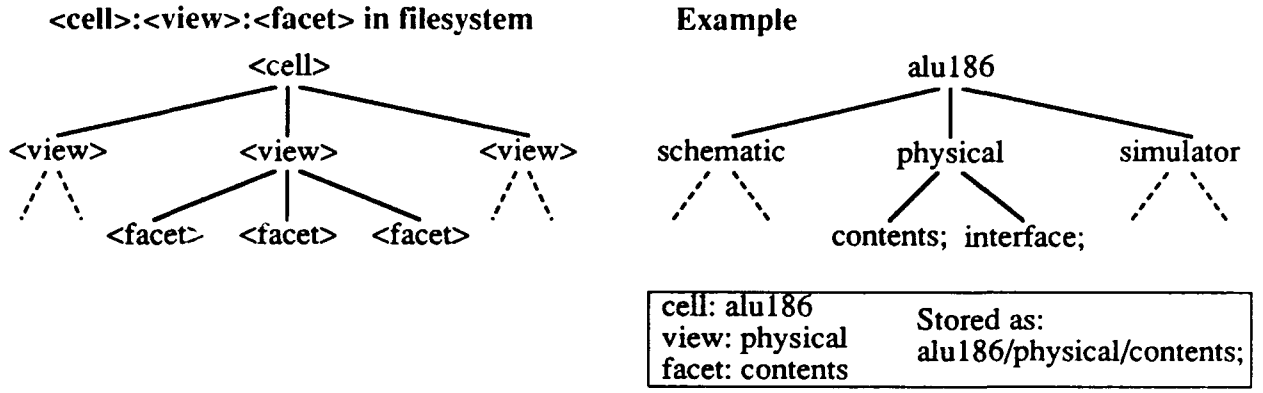



**Figure 2.6 - How facets are stored in the file system**

(instance, term, net), geometric information (box, polygon, circle, path, label, point, edge, layer), and miscellaneous information (property, bag).[2]

Relationships between octObjects are specified by their *attachments* with other octObjects. For example, a Box object may be attached to a Layer object, or a property attached to a Net or transistor Instance. Attachment of object B to object A forms a graph with a directed edge from A to B as shown in Figure 2.7. In Oct terminology, one can also say that A *contains* B (or the contents of A is B) or that the *container* of B is A. The contents of an octObject are all of the octObjects that are attached to it. The container of an octObject is the octObject that the given object is attached to.

Facets contain octObjects by the use of attachments. Figure 2.8 shows a given hierarchical Oct facet that has two instances. An example of the attachments that are created for such a design are also shown. The facet is shown to contain an Instance bag that in turn contains the

[2] Terminals are the external connection points of a cell or instance. A bag is a user-defined collection of octObjects. Properties consist of a name string and a value; they may be used to represent the dimensions for a transistor or parasitic capacitance values for nets.

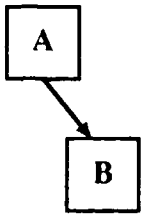



**Figure 2.7 - Attachments: octObject B is attached to octObject A**

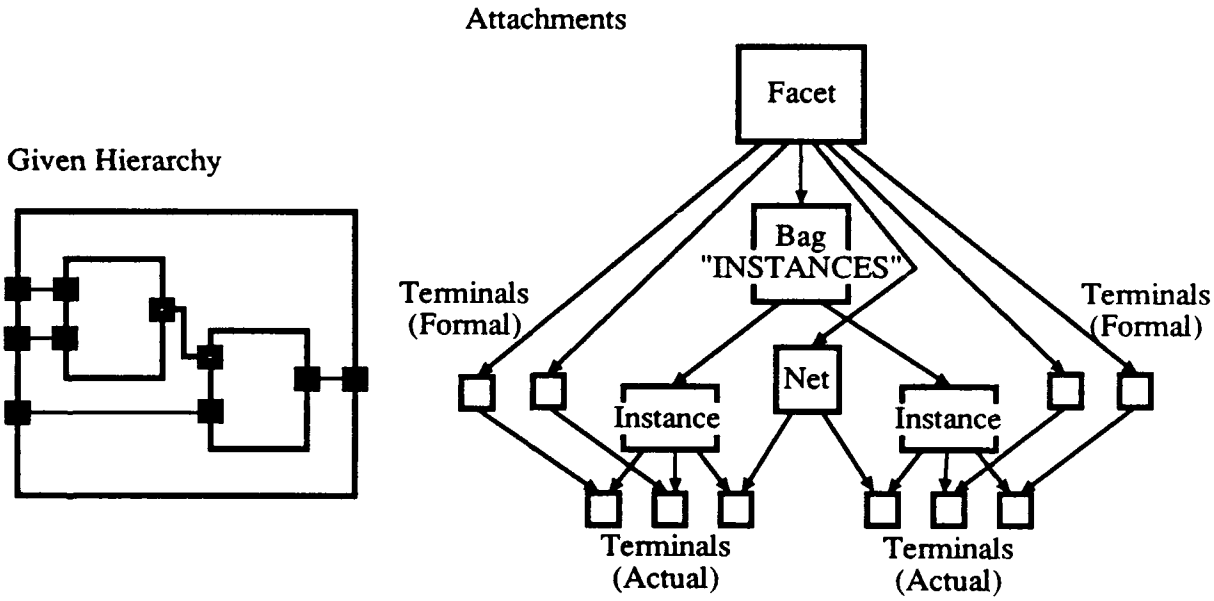



**Figure 2.8 - Simple attachments on a facet**

two instances. The single internal net is attached to the facet. The external or *formal* terminals are attached to the facet, but the so-called *actual* terminals of the instances are attached to the instances not the facet.

In Oct, one is free to make attachments in an arbitrary fashion. An Oct *policy* on a view provides the meaning or reason for the attachments in the facets of a view. In some sense it determines if an attachment makes sense, e.g., if a given attachment is "legal." For example, the policy on a facet with terminals attached to a net interprets the attachment to mean that the terminals are connected to the net. The connectivity of the design that the facet is trying to represent can be determined only by the policy for that facet. A policy can also be used to dictate where to look in the facet for specific information: the dimensions of transistor instances or the capacitance of a net, for example.

## 2.8. Oct Integration

An Oct front-end was developed for iCHARM so that it could read layout information from Oct facets. The input module uses Oct access routines to read the facet and its attached objects to subsequently build internal data structures.

### 2.8.1. Oct access routines

A program may interface with the Oct database by calling Oct access routines that read the Oct files and communicate the information within them. The Oct access routines are defined in object code libraries that must be linked with the interface program when it is compiled. The basic access routines will be described next so that a sketch of the Oct input module of iCHARM can be presented.

The declarations of the minimal set of access routines are shown in Figure 2.9. The octOpenFacet() routine must be called before any operations can be done on the "facet" argument. This routine is similar to fopen() except facets are manipulated instead of file pointers.

The octOpenMaster() routine is similar to octOpenFacet() except the "master" facet of the given "instance" is opened for processing. The routine is useful, given the "instance" facet, to get a pointer to the "master" facet that holds the original cell definition.

The octInitGenContents() and octGenerate() routines work together as shown in Figure 2.10 to *generate* all of the objects of a certain type that are attached to the "container" object. The octInitGenContents() routine is used to obtain a "generator" that is used to get all of the octObjects that are attached to the "container" that are of the type specified by "mask." The octGenerate() routine is then used in a while-loop to sequentially produce the attached objects in

```
octOpenFacet(facet)
  octObject *facet;

octOpenMaster(instance, facet)
  octObject *instance;
  octObject *facet;

octInitGenContents(container, mask, generator)
  octObject *container;
  octObjectMask mask;
  octGenerator generator;

octGenerate(generator, object)
  octGenerator generator;
  octObject object;
```

**Figure 2.9 - Selected Oct access routines**

```
octGenerator gen;
octObject net;
octObject term;

octInitGenContents(&net, OCT_TERM_MASK, &gen);
while (octGenerate(&gen, &term)) {
  /* process the terminal term */
}
```

**Figure 2.10 - Use of an Oct generator to get all attached terminals of "net"**

the "object" variable using the "generator." Similar routines exist to obtain all of the containers of an object.

### 2.8.2. Oct input module

The Oct input module is sketched in Procedure 2.2. The procedure is not as simple as the one that reads the CIF format. In CIF, the cells are defined in the file in the proper bottom-up hierarchical order so that reading them in is straightforward.

## Procedure 2.2 - Oct input procedure

```
{
  GetOctLayout()
  Input: inputarg: facet name input argument.
  Output: The internal data structures are built from Oct facet.
}
GetOctLayout(inputarg)
begin
  octBegin();
  inputfacet ← return a facet name from inputarg string;
  octOpenFacet(inputfacet);          { start at the top-level cell inputfacet }
  DefineCell(inputfacet, Cellhead);
  octEnd();
end;

{
  DefineCell()
  Input: newcell: new cell struct to build (returned).
       inputfacet: input facet.
  Output: A cell struct newcell is built from the information in inputfacet.
}
DefineCell(inputfacet, newcell)
begin
  newcell ← MakeNewCellstruct();
  GetRectsFromFacet(inputfacet, newcell);
  octInitGenContents(inputfacet, OCT_INSTANCE_MASK, instgen);
  while octGenerate(instgen, instance) returns OCT_OK do      { make instance structs for newcell }
       GetInstance(instance, newcell);
end;

{
  GetRectsFromFacet()
  Input: facet: input facet to read boxes from
       cell: cell struct being defined (to add rects to).
  Output: Oct boxes from facet are converted into rects and added to rectlist of cell
}
GetRectsFromFacet(facet, cell)
begin
  octInitGenContents(facet, OCT_LAYER_MASK, layergen);
  while octGenerate(layergen, layer) returns OCT_OK do begin          { for each layer, get all rects }
       octInitGenContents(facet, OCT_BOX_MASK, boxgen);
       while octGenerate(boxgen, box) return OCT_OK do
         add a rect to cell's rectlist from the info of box, layer;
  end;
end;
```

**Procedure 2.2 - Oct input procedure (continued)**

```
{
  GetInstance()
  Input: instance: Oct instance
      parentcell: cell struct to create instance for
  Output: an instance struct for instance has been built and inserted into parentcell.
}
GetInstance(instance, parentcell)
begin
{ get the facet that is the master (definition) of instance }
  facet ← octOpenMaster(instance);
{ instdefn is set to the cell struct that defines the instance }
  instdefn ← return the cell struct definition of facet;
  if instdefn is not already defined
  begin
        DefineCell(facet, instdefn);
  end;
{ build a new instance struct }
  newinst ← MakeNewInststruct();
  newinst→defn ← instdefn;
  copy instance's transformation matrix to newinst;
  insert newinst into parentcell's instance list;
end;
```

To read an Oct design, the hierarchy must be traversed manually. The Oct input module is a recursive traversal of the design hierarchy that starts at the top-level facet. It then visits each instance in order. If all of the instances of the current facet are defined, then the current facet is processed: the Oct boxes that are attached to the facet are read in, and instance structs are created for all of the instances. If the definition of the instance is not defined then the module calls itself again on the instance's definition facet.

# CHAPTER 3

# CIRCUIT EXTRACTION ALGORITHMS

This chapter describes the process of circuit extraction in more detail. A stated research goal of this work is to do the extraction hierarchically; hierarchical extraction is done on cells with mask-level geometries as well as subcell instances. As a first step, however, it is important to look at how flat extraction is done, that is, extraction done on cells consisting only of geometries and without any instances. Flat extraction can be thought of as the basis of hierarchical extraction since all hierarchical extractors must call a flat extractor, for example, on the leaf-level cells of a design. Hierarchical extraction is simply an extension of the basic flat procedure which then allows the extractor to handle cells with instances. Therefore, methods to carry out flat extraction will be examined first.

## 3.1. The Flat Circuit Extraction Process

As stated before, circuit extraction is the transformation of artwork information into circuit information so that one may verify that a circuit performs as intended. A flowchart of the (flat) circuit extraction process is shown in Figure 3.1.

The input to the flat extractor consists of the rectangles of the layout for each mask layer. The geometric extraction step consists of, first, the identification of transistors by finding the regions where the POLY and DIFF mask layers overlap. Second, all of the interconnection nets are identified by starting at each transistor's drain or source region and collecting all of the electrically-connected rectangles. The algorithm used and the underlying data structures are very critical to the efficiency of the geometric extraction step. The two tasks listed above are the specifics of the following general layout analysis problems:

(1) Find the intersection of geometries on two layers

(2) Find all of the connected geometries on a given layer that contain a given coordinate

Geometric extraction algorithms are compared in the next section.

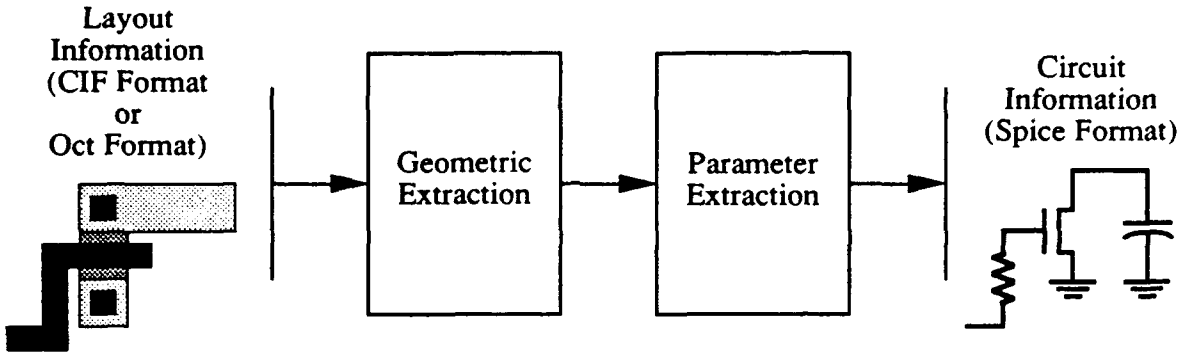



**Figure 3.1 - The circuit extraction process**

The extraction of electrical parameters is the second major step in the extraction process. In this step, the dimensions of the transistors are calculated, and the resistance and capacitance of the interconnection nets are calculated. The final output of the extractor consists of the transistors extracted and the parasitic resistance and capacitance of the interconnects.

The extraction of the parasitic values, as shown in Figure 3.2, is actually a *modeling* step. Each rectangle of a net is considered to form a parallel-plate capacitance with the chip substrate which is the ground plane for the chip. A simple model is used to calculate this *substrate* capacitance; the capacitance is calculated by finding the total area and perimeter of the rectangles that make up the net, multiplied by a technology-dependent constant.

Generalized resistance extraction of arbitrary combinations of rectangles is not as easy a problem as capacitance estimation. For the resistance of each interconnection net, the rectangles that make up the net are split into simpler, nonoverlapping pieces that cover the same area as the original rectangles. The total resistance across the length of the rectangles that make up the net is the sum of the resistances through each of the simplified pieces. Parasitic extraction is covered in detail in Section 4.2.

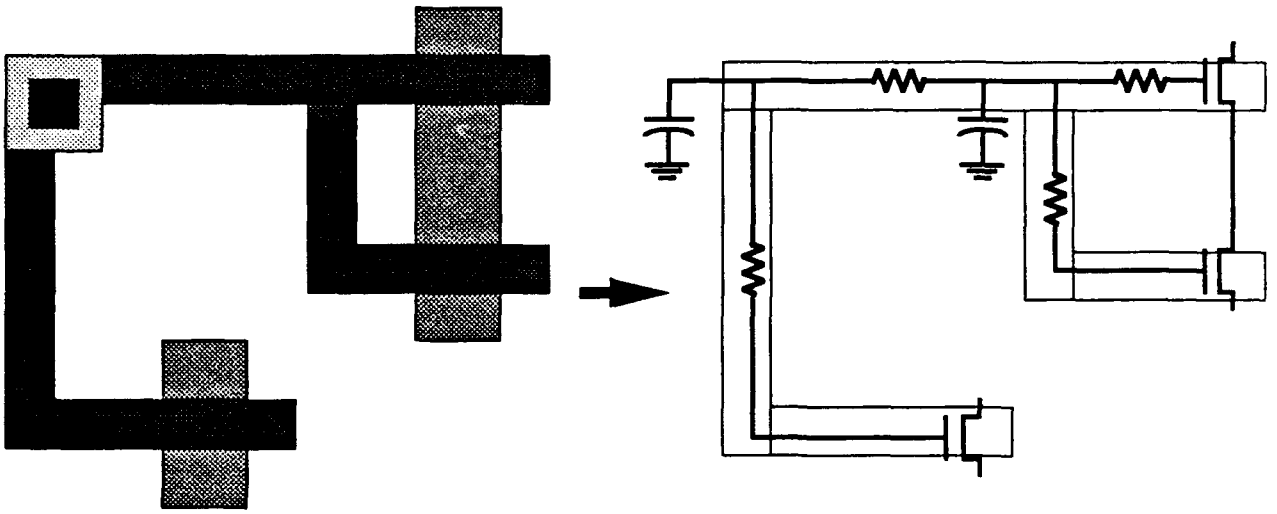


**Figure 3.2 - Parasitic extraction of interconnection nets**

## 3.2. Geometric Extraction Algorithms

An input layout for a typical circuit is made up of a large number of rectangles. The geometric extraction step, which is the basis of the entire extraction process, consists of a number of operations on each of the input rectangles.

The data structure chosen to store and retrieve the layout rectangles is crucial to the efficiency of the algorithm used to carry out the geometric extraction step. In addition to the time complexity of the algorithm, the total memory usage of the process must be considered. Even in systems with a large virtual memory, the thrashing caused by a small physical memory when the memory image is large can significantly degrade the performance of an algorithm. In this section, several layout data structures that have been employed in previous extractors are discussed along with the one that is used in iCHARM.

### 3.2.1. K-D trees

The first technique uses multi-dimensional binary trees or so-called *K*-dimensional (*K*-D) trees to store the input rectangles [13], [15]. This structure has been shown to be particularly

efficient for a layout editor although it was originally developed for the general problem of associative retrieval of data records with a multiple number, $K$, of *keys* or attributes. For the purpose of storing rectangles, $K = 4$, with the four keys, $k_i = \{X_{min}, Y_{min}, X_{max}, Y_{max}\}$ for $i = \{0,1,2,3\}$, representing the bounds of each rectangle.

Each node in a 4-D tree corresponds to a rectangle in the layout. At level $l$ in the tree, the value $i$ is defined as the *discriminator* of a node at level $l$. At any node in the tree, the coordinate $k_i$ of the rectangle is used to bisect the layout plane, and any subsequent rectangles that are inserted into the tree are placed in the node's right or left subtree depending on where the new rectangle lies in relation to the bisection line. In other words, for a node $t$, the key values $k_i$ of each node in the left subtree of $t$ are less than the $k_i$ of $t$, and the key values $k_i$ of each node in the right subtree of $t$ are greater that the $k_i$ of $t$.

Figure 3.3 shows a 4-D tree created from the given layout as an example. In each node, the circled key value is used to split the input plane for the right and left subtrees. For rectangle A, the $X_{min}$ (left edge) value of 3 is used to split the layout plane: all rectangles to the left of $x = 3$, namely, B, D, E, and H are put in the left subtree of A, and rectangles C, F, G, and I have an $X_{min}$ larger than 3 so they are put into the right subtree of A. For rectangles with a discriminator of 1 (B and C), the bottom edge is used to split the input plane; for discriminator 2 rectangles (D, E, F, G), the right edge is used, etc.

The first step in geometric extraction is the identification of all transistors, that is, where POLY and DIFF rectangles intersect. This is done by finding, for each POLY rectangle, the DIFF rectangles that intersect it. Let $N$ be the total number of input rectangles. The number of POLY rectangles is $O(N)$ and Rosenberg [13] reports that *intersection search*, in which all rectangles that intersect a given region are found, takes $O(\log N)$ for $K$-D trees. Therefore, transistors may be identified in $O(N \log N)$.

To identify all of the connected rectangles in the geometric extraction step (net identification), Marple [8] reports that $K$-D trees should take $O(n \log N)$ time, where $n$ is the number of rectangles in the net. Since $n$ is $O(N)$, the entire geometric extraction step with $K$-D trees should take $O(N \log N)$ time. Of course, all $N$ records need to be kept in memory for the extraction process, so the space complexity is $O(N)$.

The $K$-D trees have a number of disadvantages, the primary one being that certain insertion orders will create an unbalanced tree that degenerates into a simple linked list. The running time

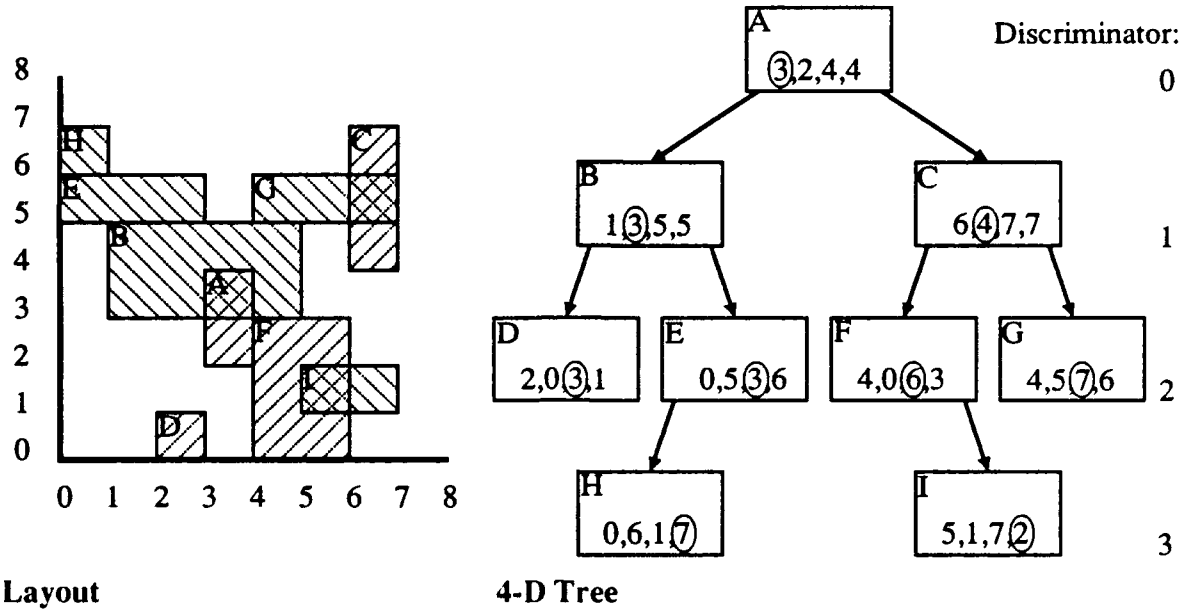


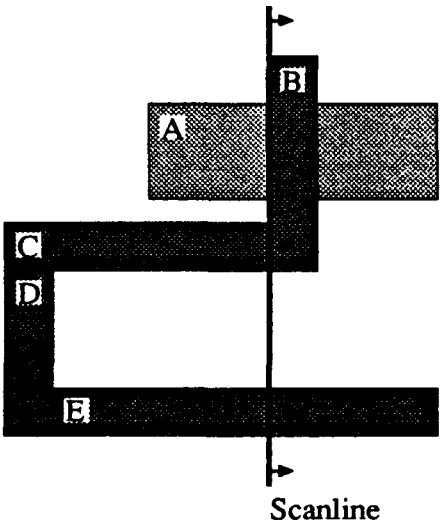

**Figure 3.3 - A layout and its corresponding 4-D tree**

for the structure will degrade in this case so additional processing must be done when rectangles are added or deleted from the tree to keep the tree balanced.

### 3.2.2. Scanline algorithms

Rather than consider all of the rectangles at once, in a scanline extraction algorithm, the rectangles are considered only when they cross a vertical *scanline* that sweeps across the input plane from left to right [1], [4], [16]. The scanline stops at points in the plane where the rectangles begin and end.

At each stopping point, all rectangles that are beginning (i.e., the scanline is at leftmost edge of the rectangle) are inserted into the "scanline" data structure; all rectangles that are ending are deleted from the scanline structure. The scanline structure can be thought to contain the "active" rectangles for the extraction process. When a rectangle is added to the scanline structure, it is determined if it intersects with any other rectangle currently in the structure.




**New Rects (to be added to scanline):** B

**Old Rects (to be deleted from scanline):** C

**Existing Rects in scanline:** A, C, E

**Nets created:** Two {(C, D, E), (A)}

Rect B will be added to the net (C, D, E) and a transistor will be created

**Figure 3.4 - The scanline extraction process**

Intersections of POLY and DIFF rectangles create transistors. On the other hand, if two intersecting rectangles are on layers that electrically connect, then the new rectangle is added to the list of rectangles that make up a net, of which the rectangle already in the scanline structure is a member.

Figure 3.4 shows an example of extraction in progress that uses a scanline. Rect A is a DIFF rect, and all other rects are POLY. At present the scanline is positioned to add rect B to the scanline structure since its left edge is on the scanline. When it does so, rect B will be added to the net that already contains rects C, D, and E. The intersection of rects A and B will create a transistor, and finally at the end of this scanline position, rect C will be deleted from the scanline structure and no longer be considered active.

Actually, the scanline technique uses simple linked lists as the data structure to store the rectangles, although the lists are kept sorted. In this case, the scanline technique is more important than the data structure used to store the rectangles. Scanlines are useful for batch applications like extraction, but since the underlying data structure is a linked list, using scanlines

would not be advisable for the interactive queries or random deletions of rectangles that would be frequent in layout editing.

However, Szymanski [16] reports the time complexity of geometric extraction with scan-lines should also be $O(N \log N)$ with an expected space complexity of only $O(\sqrt{N})$. The space complexity is dictated by the maximum number of rectangles cut by a vertical line in the layout plane, which has an expected value for most designs of $O(\sqrt{N})$.

### 3.2.3. Corner stitching

Corner-stitching has emerged as a successful data structure for storing layout rectangles to implement such tools as a layout editor, design rule checker, compactor, and extractor [14], [8]. In corner-stitching, both the layout rectangles and the empty space between them are stored with nonoverlapping *tiles*. Input rectangles that overlap are handled by creating special overlap tiles or by combining them with existing tiles to create larger nonoverlapping tiles. Two *stitches* per tile are used to connect adjacent tiles (see Figure 3.5 for an example of a corner-stitched layout). Because of this, the most efficient operation for corner-stitching is nearest neighbor searching.

Marple [8] reports that corner-stitching is slower than $K$-D trees for insertion and deletion operations. Therefore, $K$-D trees are favored for fast interactive layout editing with a large number of rectangles, and corner-stitching is advocated for batch processing such as that involved in extraction.

Net identification using the "node search" algorithm in Marple [8] is reported to take $O(n)$, where $n$ is the number of rectangles in the net. Transistor identification uses the "area search" algorithm to find the intersection of POLY and DIFF that creates each transistor; then the node search procedure can be used to combine transistors made up of several tiles. Area search is reported to be $O(n)$ also. Therefore, the time complexity of the geometric extraction step with corner-stitching is linear.

The primary disadvantage of corner-stitching has been reported to be the difficulties it has in handling a large number of different layers [8] and multiple overlapping regions [13].

### 3.2.4. The geometric extraction algorithm in iCHARM

The choice for the geometric extraction algorithm used in iCHARM was perhaps determined more by practical concerns than by theoretical advantages. Two existing extractors that were developed at the University of Illinois were evaluated with the intent to use the geometric

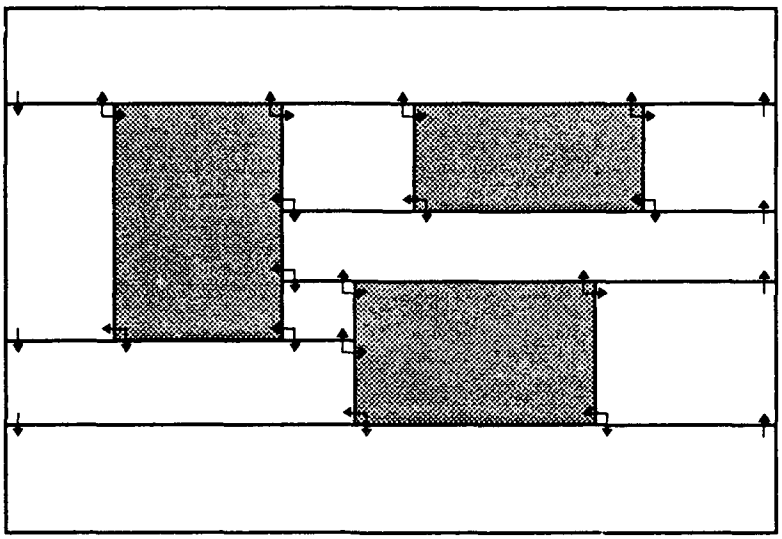


**Figure 3.5 - A corner-stitched layout**

extraction code in one of the existing implementations to develop a new hierarchical extractor, one that is integrated with the Oct/Vem system. It was felt that nothing would be gained from starting from scratch. The first extractor, iCPEX, uses 4-D trees to store the input rectangles [15], and the second extractor, PACE, uses a scanline extraction algorithm [1].

The extractor in the Magic layout editor was not evaluated, primarily because hierarchical extraction is already implemented in the program. As will be shown in Chapter 5, the Magic extractor handles a difficult hierarchical test-case by flattening the hierarchy. A new technique was developed for iCHARM that extracts the test-case without flattening. In addition, it was felt that the large size of the Magic program would make it difficult to learn and to add new features to it. The iCPEX and PACE programs were of a more manageable size.

After extensive study, the PACE program was found to be a better implementation. The next chapter presents the details of the iCHARM implementation which uses code from the PACE program to do the basic flat extraction.

# CHAPTER 4

# iCHARM FLAT EXTRACTION

This chapter documents the implementation of iCHARM to do geometric extraction, power bus extraction, and parasitic extraction. Since only basic techniques are covered, hierarchy is not yet utilized in this analysis. As mentioned before, a scanline algorithm is used to find the geometric relationships between the mask-level rectangles.

## 4.1. Geometric Flat Extraction

Geometric extraction, in which the basic connectivity of a cell is formed, is the first step in the extraction process. The geometric extraction subroutine takes a cell as its input argument; the rectangles in the *rectlist* of the cell are used as input to create the nets and transistors that are output in the *netlist* and *tranlist* fields of the cell struct.

The extraction is considered flat because only rectangles are used in the geometric extraction process. If the input cell originally contained instances (in other words, hierarchy), then the cell must be flattened using the Flatten() procedure shown in Procedure 2.1 before flat geometric extraction is performed. The major steps in the flat extraction process are the following:

```
{
  FlatExtract()
  Input: cell: The cell to extract. Input rectangles are in cell's rectlist.
  Output: The extracted net and transistors in cell's netlist and tranlist.
}
FlatExtract(cell)
begin
  if cell has hierarchy then
        Flatten(cell);
  GeometricExtract(cell);
  ParasiticExtract(cell);
  report transistors and parasitics for cell;
end;
```

### 4.1.1. Geometric data structures

The C-language declarations for the rect, linkel (link element), net, and tran (transistor) structures are introduced in Figure 4.1 to permit an in-depth presentation of the geometric extraction procedure. Each struct has a *nextptr* field that is used to make a linked-list of each type.

Each electrical net is defined by a net struct. The net is identified by its *netnum*, which is generated internally by iCHARM, or the user may name the net by defining a label at the location

```
/* NET DEFINITION */
typedef struct _net {
   long netnum;                  /* number of the net */
   char *lab;                    /* label for the net */
   union {
         char *ptr;              /* a scratch pointer field */
         int cnt;                /* a scratch count field */
   } scratch;
   struct _rect *rectlist;       /* list of rects for this net */
   int rcount;                   /* count of number of rects in 'rectlist' */
   struct _term *termlist;       /* list of terminals for this net */
   int tcount;                   /* while the net is in the scanline struct,
                                    count of terms in cell's input termlist */
   float cap;                    /* the capacitance of the net */
   struct _net *nextptr;
} net, *netptr;

/* MASK RECTANGLE */
typedef struct _rect {
   long coord[4];                /* xmin,ymin,xmax,ymax */
   char layer;                   /* layer number */
   union {
         struct _net *setptr;    /* ptr to owner net (rect in netp→rectlist) */
         struct _linkel *tranpt; /* ptr to transistors (drain/source rect) */
         char *prirect;          /* ptr to primary rect or term (in scanline) */
         struct _tran *settran;  /* ptr to transistor (channel rect) */
   } un;
   struct _branchnode *bnode;    /* rect's branchnode (parameter extraction) */
   struct _rect *nextptr;
} rect, *rectptr;
```

**Figure 4.1 - Data structures for geometric extraction**

of a rectangle of the net. The *lab* string is the user name for the net. After the geometric extraction process is over, the *rectlist* field contains a list of rectangles that make up the net.

The rectangle struct includes the $\{X_{min}, Y_{min}, X_{max}, Y_{max}\}$ bounds of the rectangle and the layer it is on. The *un* union field is used for temporary pointers during the extraction process.

The linkel (link element) struct is used to hold a list of pointers to arbitrary items: the pointer *lptr* is cast to the type of object that it points to.

The tran struct defines each transistor. The *chanptr* field points to a list of channel rectangles which are the rectangles that make up the channel region of the transistor. The *pdiff* field is

```
/* L I N K E L - a general structure for a list of pointers */
typedef struct _linkel {
   char *lptr;
   struct _linkel *nextptr;
} linkel, *linkelptr;

/* T R A N S I S T O R */
typedef struct _tran {
   long trannum;                    /* the transistor number */
   struct _rect *chanptr;           /* a list of channel rectangles */
   char ttype;                      /* type of the transistor */
   union {
         struct _linkel
                  *pdlink[2];       /* the poly and diff links */
         struct {
            long area;              /* area of the channel rectangles */
            long length;            /* length of intersecting diff rectangles */
         } al;
   } pdiff;
   long gnode, snode, dnode;        /* gate, drain, and source node numbers */
   long gnetnum, snetnum,/* gate, source, and drain net numbers */
          dnetnum;
   union {
         char *ptr;                 /* a scratch pointer field */
         int cnt;                   /* a scratch count field */
   } scratch;
   struct _tran *nextptr;
} tran, *tranptr;
```

**Figure 4.1 - Data structures for geometric extraction (continued)**

used for temporary storage during extraction. The *dnetnum*, *gnetnum*, and *snetnum* fields are the *net* numbers of the drain, gate, and source terminals of the transistor. For example if a transistor's gate is connected to a net with a *netnum* of "57," then the *gnetnum* of the transistor is set to "57."

On the other hand, the *dnode*, *gnode*, and *snode* fields are the *node* numbers of the drain, gate, and source. In geometric extraction, all of the connected rectangles are collected into one *net* and all of the rectangles are considered to be equipotential. In parasitic resistance extraction, each net is split up into equipotential regions with resistances between them; these equipotential regions are defined as *nodes*.

### 4.1.2. Basic scanline extraction procedure

A general scanline procedure is presented in Procedure 4.1. The procedure processes the rectangles in *rectlist*. Since the scanline proceeds from left to right, the *rectlist* must be sorted. The *curscan* variable keeps the current scanline position. The rectangles are removed from the *rectlist* in sorted order. The scanline "stops" at the $X_{min}$ coordinate of the rect at the head of *rectlist* and processes all rects that have the same $X_{min}$ coordinate.

Rects are kept in the scanline structure while *curscan* $\geq$ the rect's $X_{min}$ and *curscan* $\leq$ the rect's $X_{max}$. In iCHARM, the scanline structure is implemented as a simple linked list. More sophisticated data structures could be used to support faster deletion or searching of rectangles in the scanline structure.

Procedure 4.1 will now be discussed in detail. At each stopping point of the scanline, *curscan* is set to the left edge of the rectangle at the head of *rectlist*. Next, the scanline structure is searched to find the rects already in the scanline structure that have been passed by *curscan* and must be deleted. This is accomplished in the DeleteFromScanline() routine, which also may do some algorithm-specific processing of the rectangle that is to be deleted from the scanline structure.

In the next step, *prect*, the rect with its $X_{min}$ edge at *curscan*, is taken out of *rectlist* so that it may be inserted into the scanline structure. The AddToScanline() routine is called to perform any algorithm-specific processing of the new rectangle. The input rect *prect* is checked against all rects in the scanline structure to see if it touches any of them. The touching rects and *prect* are processed in various ways according the purpose of the scanline procedure being implemented. Then *prect* is added to the scanline structure.

**Procedure 4.1 - A generic scanline procedure**

```
{
  ScanlineAlg()
  Input: rectlist: list of rectangles to process.
  Output: The rects have been processed (from left to right) in some way.
}
ScanlineAlg(rectlist)
begin
  sort rectlist in increasing order of X_min;
  while there are rects in rectlist do begin
        curscan ← X_min of head of rectlist;
        DeleteFromScanline();
        for all rects, prect, with X_min = curscan do begin
          prect ← Pop(rectlist);
          AddToScanline(prect);
        end;
  end;
  curscan ← +∞;
  DeleteFromScanline();          { delete (and process) all the remaining rects from scanline }
end;

{
  AddToScanline()
  Input: prect: the rectangle to process and add to scanline.
  Output: The rects in the scanline that touch prect have been processed in some way,
          and prect has been added to the scanline structure.
}
AddToScanline(prect)
begin
  for all rects, srect, in scanline that touch prect do   { find all rects in scanline that touch prect }
        process interaction of srect and prect;
  add prect to scanline;
end;

{
  DeleteFromScanline()
  Output: Rects past curscan have been processed in some way and deleted from the scanline structure.
}
DeleteFromScanline()
begin
  for all rects, srect, with Y_max < curscan do begin   { flush all rects in scanline past curscan }
        srect ← Pop(scanline);
        process outgoing srect if necessary;
  end;
end;
```

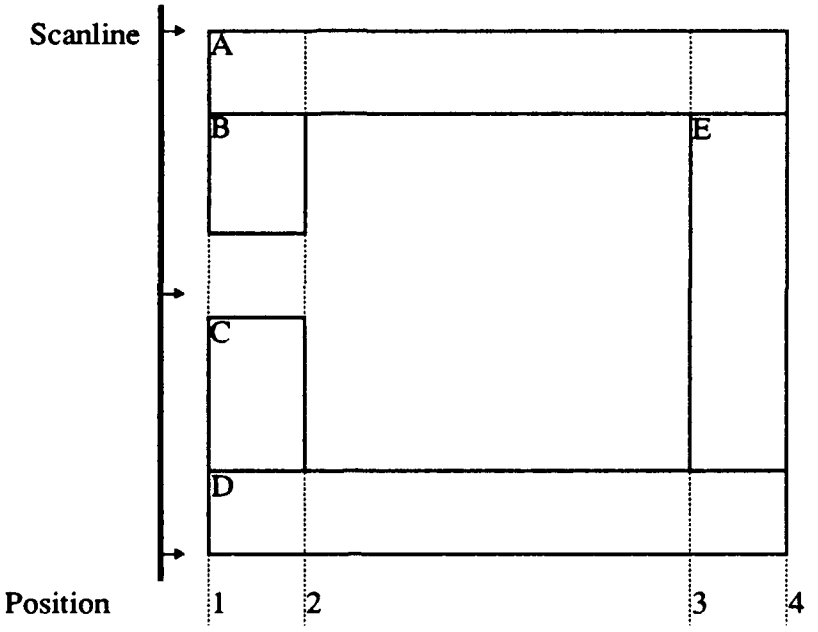



**Figure 4.2 - A collection of rectangles to process using a scanline**

Figure 4.2 shows a collection of rectangles to be processed by a scanline algorithm. The scanline will stop at points 1, 2, 3, and 4. At each point, the following will occur:

At point 1:

Rects A, B, C, and D will be added to the scanline structure. Let A be added first. Then when B is added, the interaction between A and B will be considered and processed. Since C does not touch A and B, rect C is simply added to the scanline structure. When D is added, the C-D interaction is processed.

At point 2:

No new rects are added to the scanline structure. Rects B and C are deleted from the scanline.

At point 3:

Rect E is added to the scanline structure. A and D are still in the scanline, so the interaction of E-A and E-D is processed before E is added.

At point 4:

All remaining rects are deleted from the scanline.

The next sections will present the application of the basic scanline procedure to implement geometric extraction in iCHARM.

### 4.1.3. Scanline net extraction

First, the problem of finding all the nets of a cell, which amounts to partitioning the input list of rectangles for the cell into electrically-connected groups, is considered. Since each group constitutes a net, the output of the net extraction procedure is a list of net structs, and each net has a list of rectangles for the net.

The net extraction procedure given in Procedure 4.2 is an extension of the basic scanline procedure. The basic procedure is modified by filling in the AddToScanline() and DeleteFrom-Scanline() routines. In the AddToScanline() routine, CreateNet() is called to make a new net struct for the rect *prect*. The output of CreateNet() is shown in Figure 4.3. In CreateNet(), a copy of *prect* is inserted into the new net's rectlist and the pointer *netrect* is returned by the routine. The *netrect*'s un.setptr is set to point back to the net struct.

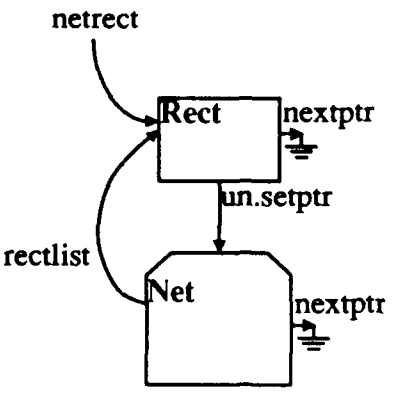



**Figure 4.3 - The output of the CreateNet() routine**

## Procedure 4.2 - Net extraction procedure using a scanline algorithm

```
{
  NetExtract()
  Input: cell: contains a list of rectangles to process.
  Output: The cell's netlist has been created.
}
NetExtract(cell)
begin
  ScanlineAlg( cell's rectlist );
end;

{
  AddToScanline()
  Input: prect: the rectangle to process and add to scanline.
  Output: A net struct is created for prect, the nets of the rects in the scanline that
          touch prect are combined, and prect is added to the scanline structure.
}
AddToScanline(prect)
begin
  netrect ← CreateNet(prect);
{ find all rects already in scanline that touch prect }
  for all rects, srect, in scanline that touch prect do
  begin
      combine nets that contain srect and prect;
  end;
  add prect to scanline;
end;

{
  DeleteFromScanline()
  Output: All rects past curscan are deleted from the scanline structure;
          complete nets are added to the cell's netlist.
}
DeleteFromScanline()
begin
{ flush all rects in scanline past curscan }
  for all rects, srect, with Xmax < curscan do
  begin
      srect ← Pop(scanline);
  { "complete" nets have no rects in the scanline, i.e., all of the rects that make up the net
    are past curscan }
      if the net of srect is "complete," then add the net to the cell's netlist;
  end;
end;
```

After CreateNet(), the nets of the rects in the scanline structure that touch *prect* are combined with *prect*'s net since all connected rects should be in a single net. This step corresponds to "processing the interaction of *srect* and *prect*" in the generic procedure of Procedure 4.1. Subsequently, *prect* is added to the scanline structure: *prect* is added to the *scanline* list, and the un.prirect field of *prect* is set to point to *netrect*.

DeleteFromScanline() is also modified for net extraction. If a rect is to be deleted from the scanline and it is the last rect in the scanline structure for its net, then the net is inserted into the *cell*'s netlist. This step corresponds to "processes the outgoing *srect*" in Procedure 4.1.

The net extraction procedure is illustrated in Figure 4.4 using the same rectangles of Figure 4.2. The scanline contents are shown at each value of *curscan* in Figure 4.4.

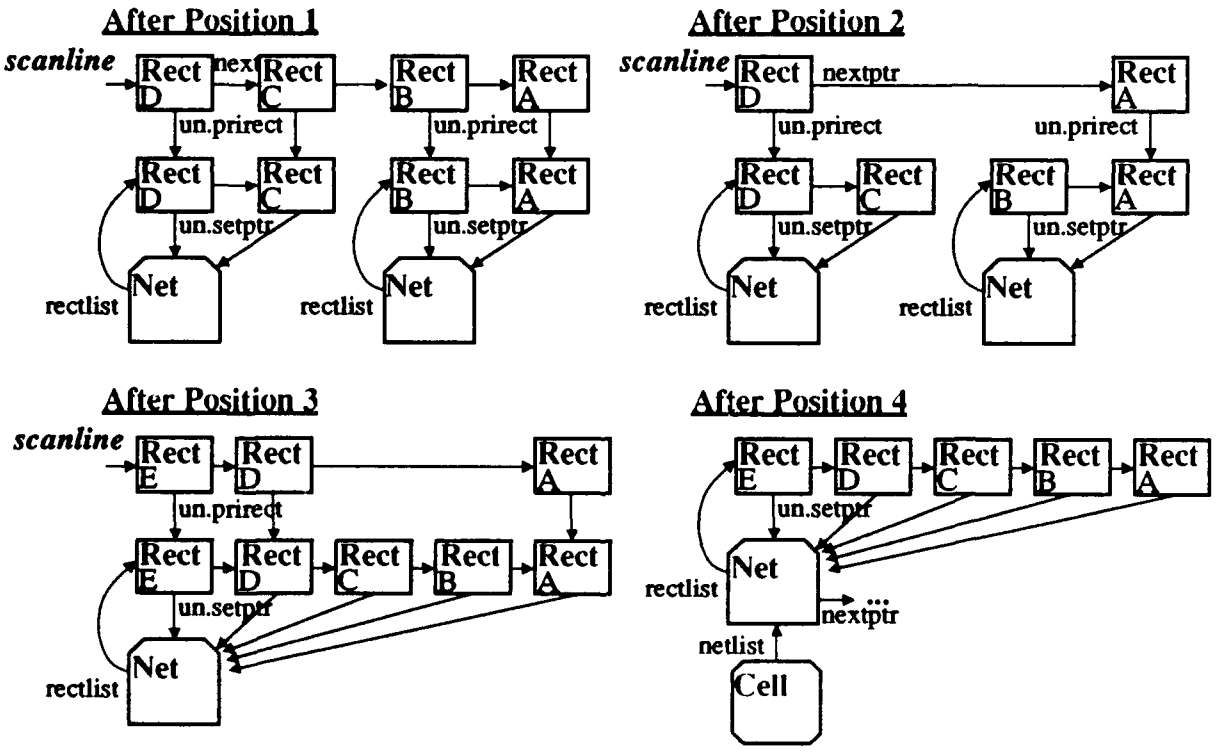



**Figure 4.4 - The scanline contents for the net extraction of Figure 4.2**

At position 1, rects A, B, C, and D are inserted into the scanline structure. Let the insertion order be A, C, B, then D. Rect A will create one net, as will rect C (from CreateNet()). Rect B will create another net, then combine nets with A (both A and B are put in the same rectlist) since A and B touch. Similarly, rect D will create a new net, then combine with C. Two nets finally remain. At position 2, rects B and C are deleted from the scanline structure. At position 3, rect E is added to the scanline and combines both nets since it joins rects A and D. Finally, at position 4, all rects are deleted from the scanline so the net is added to the cell's *netlist*.

### 4.1.4. Scanline transistor extraction

The scanline procedure that extracts the nets may be modified to identify transistors. The AddToScanline() procedure of Procedure 4.2 should be modified as shown in Procedure 4.2a. The CreateTransistor() routine creates a new transistor structure. In this routine, the intersection area of the POLY and DIFF rectangles is used to create a channel rectangle that is inserted in the new transistor's *chanptr* field. Pointers to the rectangles that form the transistor's drain, source,

---

**Procedure 4.2a - Transistor creation procedure using AddToScanline()**

```
{
  AddToScanline()
  Input: prect: the rectangle to process and add to scanline.
  Output: A net struct is created for prect, the nets of the rects in the scanline that touch prect are
      combined, and prect is added to the scanline structure. A transistor is created if prect
      and any rects in the scanline form one.
}
AddToScanline(prect)
begin
  netrect ← CreateNet(prect);
{ find all rects already in scanline that touch prect }
  for all rects, srect, in scanline that touch prect do begin
      if prect and srect form a transistor then
        CreateTransistor(prect, srect);
      else
        combine nets that contain srect and prect;
  end;
  add prect to scanline;
end;
```

---

and gate terminals are recorded in the transistor's *pdiff* field. When the rectangles of the transistor are beyond the *curscan* position, the transistor is output to the cell's transistor list (*tranlist*).

The modified net extraction procedure that also creates transistors accomplishes the majority of the geometric extraction step. In iCHARM, a subroutine named processtran() is called after the nets and transistors are identified to do some final processing of the data structures. The processtran() routine sets the drain, gate, and source net and node numbers for each transistor. It calculates the channel area and length of each transistor and stores it in the *pdiff.al* field. The routine also rearranges the pointers to the drain, source, and gate rectangles so that the pointers point back to the transistor from the rectangles.

Figure 4.5 shows the data structures that are created for a simple layout after geometric extraction is complete. The drain, source, and gate rects use their *un.tranpt* field (through linkel

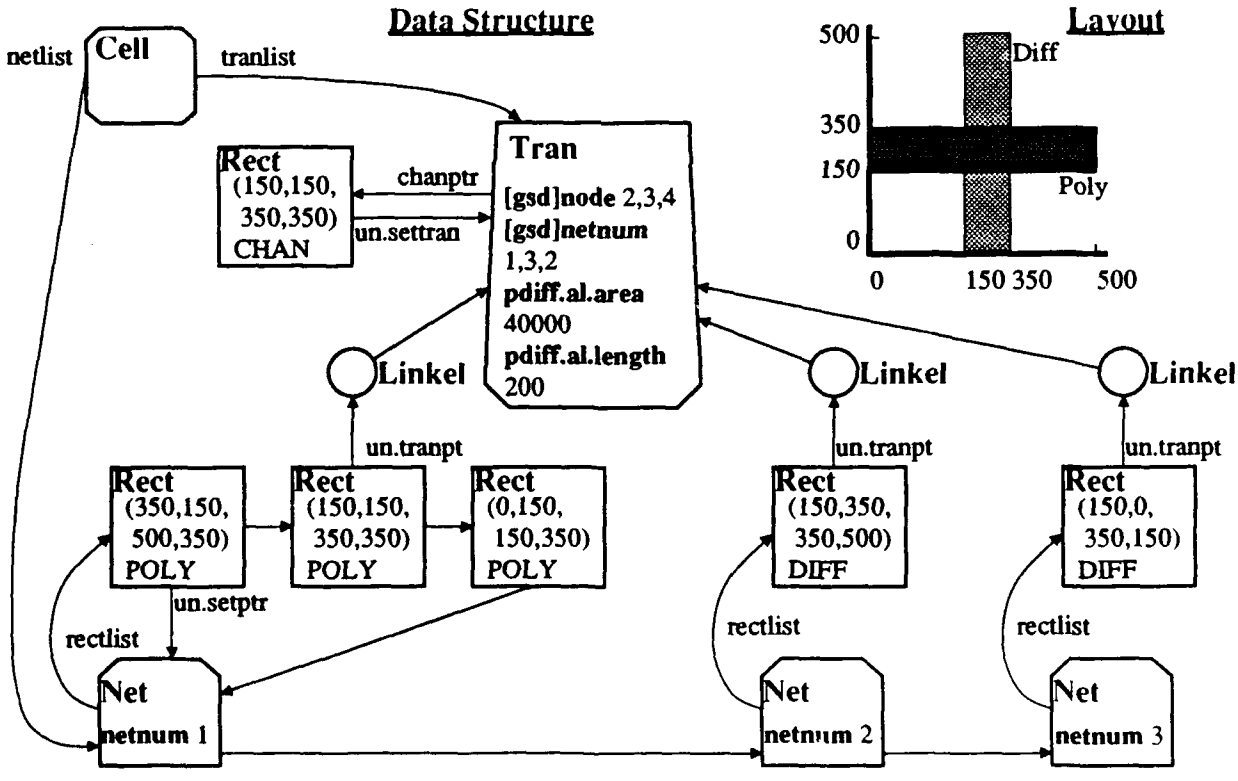



**Figure 4.5 - The data structures created after geometric extraction**

structs) to point to the transistor that is connected to the rects. Notice in Figure 4.5 that the *netnum*'s of the nets are used for the transistor's *dnetnum*, *gnetnum*, and *snetnum*.

This final data structure is then passed to the parasitic extraction module where the rectangles in each net's *rectlist* are used to compute parasitic resistance and capacitance values.

## 4.2. Parasitic Flat Extraction

Once the rectangles of each net are collected, the parasitic resistance and capacitance values of each net can be computed. The input to the parasitic extraction module is the rectangles of each net, and the rectangles are used to form a distributed RC circuit for each net.

As shown in Figure 4.6, there are two parasitic extraction modes in iCHARM; the first extracts only parasitic capacitances for the rectangles of a net. All of the rectangles of the net are considered equipotential and each rectangle is modeled as a single capacitance to ground. The total capacitance of the net is the sum of all of the capacitances for each rectangle.

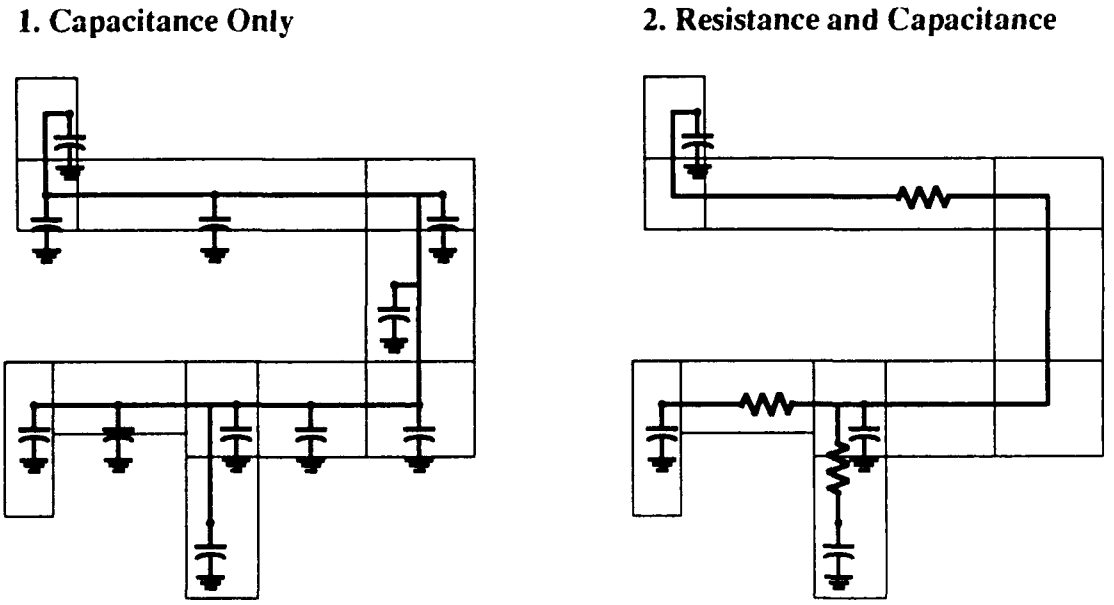



**Figure 4.6 - Two modes of parasitic extraction**

The second parasitic mode extracts resistances and substrate capacitances for each net. In so doing, the net is modeled as a distributed RC circuit and all of the rectangles of the net are no longer considered at the same potential. Within each net, electrical nodes are formed with resistances between them. This is the basis for the difference between a "net" and a "node" in iCHARM's terminology.

As one can imagine, resistance extraction is more computationally demanding than capacitance extraction. Furthermore, the addition of resistances to the extracted circuit increases the number of electrical nodes that must be simulated and can lead to complex RC networks. In most cases, sufficient simulation accuracy may be attained with only capacitance extraction; in the cases when resistances need to be extracted, the RC network can be approximated by a smaller network by a node reduction phase that is featured in iCPEX [15].

### 4.2.1. Data structures for parasitic extraction

Two data structures are used by iCHARM for parasitic extraction: the branchnode and edge structs that are shown in Figure 4.7.

```
/* B R A N C H N O D E - a branchnode struct */
typedef struct _branchnode {
   long nodenum;              /* the number of the node */
   boolean marked;            /* a flag used in various algorithms */
   char branchtype;           /* BRANCH, KNOT, UNKNOWN, or NOTTOUCH */
   char port;                 /* is POLYPORT, DIFFPORT or NOTPORT */
   int numconn;               /* number of touching rects, transistors */
   struct _rect *chanptr;     /* used if DIFFPORT, gives chan rects */
   struct _edgenode *edges;   /* edges to connecting branch nodes */
   float cap;                 /* the capacitance of the node */
} branchnode, *branchnodeptr;

/* E D G E N O D E - an edge struct */
typedef struct _edgenode {
   struct _rect *rect2;       /* pointers to the other rect */
   float cap, res;            /* capacitance, resistance */
   boolean marked;            /* a bit field used in algs */
   struct _edgenode *nextptr;
} edgenode, *edgenodeptr;
```

**Figure 4.7 - Branchnode and edge data structures**

A basic operation performed for both capacitance and resistance extraction is the creation of an "electrical connectivity graph" (the terminology is from Belkhale [2]) from the rectangles of the net. Each rectangle is a node or vertex in the graph, and edges are created from each rectangle (branchnode) to each of its adjacent rectangles. The electrical connectivity graph is used to calculate the capacitance of each rectangle and to model the rectangles as an RC network.

The electrical connectivity graph can be easily constructed by the scanline algorithm shown in Procedure 4.3. To create a graph from a collection of rectangles, first a branchnode struct is created for each rectangle (it is attached to the *bnode* pointer of the rectangle). The rectangles of

---

**Procedure 4.3 - Electrical connectivity graph creation procedure**

```
{
  CreateGraph()
  Input: net: contains a list of rectangles to process.
  Output: The net's electrical connectivity graph has been created.
}
CreateGraph(net)
begin
  for all rects, prect, in the net's rectlist;
        add branchnode to prect's bnode field;
  ScanlineAlg( net's rectlist );
end;

{
  AddToScanline()
  Input: prect: the rectangle to process and add to scanline.
  Output: A net struct is created for prect, the nets of the rects in the scanline that touch prect
          are combined, and prect is added to the scanline structure.
}
AddToScanline(prect)
begin
{ find all rects already in scanline that touch prect }
  for all rects, srect, in scanline that touch prect do
  begin
        if srect and prect are on the same layer then
          make an edge from prect to srect and vice versa;
  end;
  add prect to scanline;
end;
```

---

the net are run through a scanline algorithm with the AddToScanline() procedure, as shown in Procedure 4.3, in which an edge struct is made for each adjacent rectangle. The end result is that each rectangle can access all of its adjacent rectangles by going through its list of edges in the *edges* field of the branchnode. Figure 4.8 shows a set of rectangles, its connectivity graph, and the branchnode/edge structure created for it.

### 4.2.2. Capacitance extraction

The parasitic capacitance of a conductor process is made up of the conductor's capacitance with respect to the chip substrate as well as with other neighboring conductors. The former capacitance is called the *substrate* or self-capacitance, and the latter is called the *coupling*

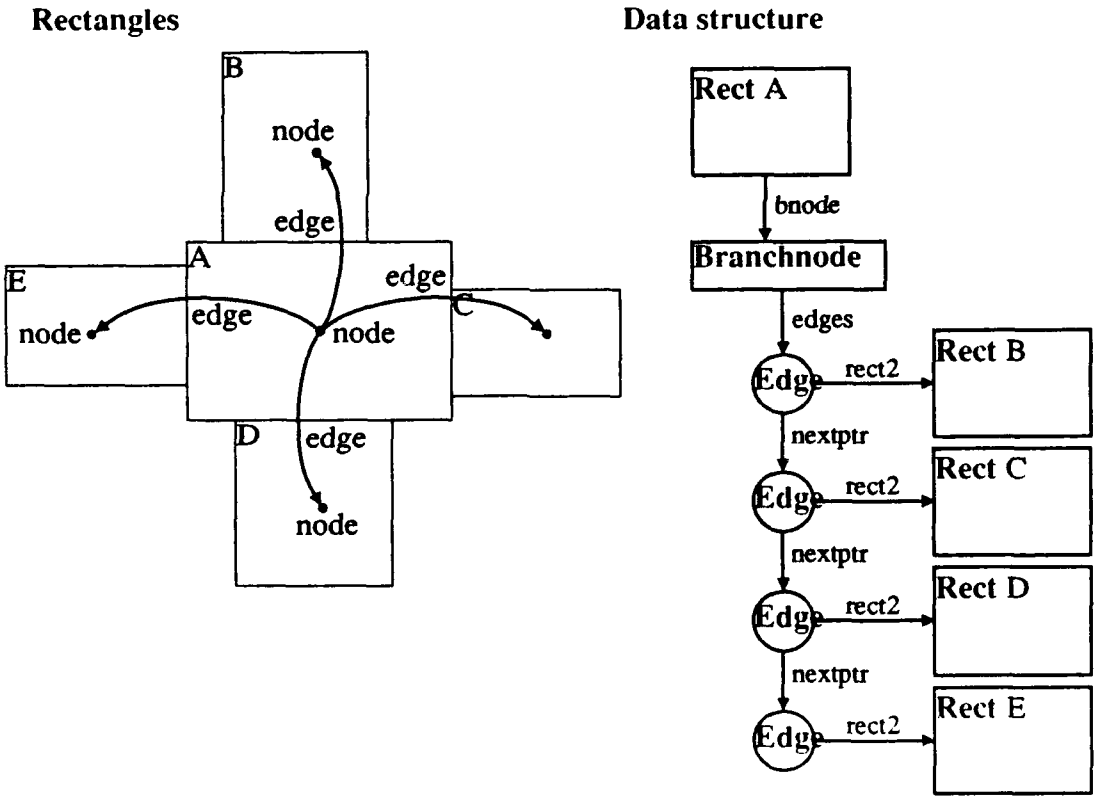



**Figure 4.8 - Electrical connectivity graph example**

capacitance. In iCHARM, the substrate capacitance of each rectangle is calculated by modeling it as a single parallel-plate capacitor to ground. The capacitance can be accurately modeled by the following equation which uses the area and perimeter of the rectangle:

$$Cap_{rect} = area_{rect} \times K_{area}[layer_{rect}] + perim_{rect} \times K_{perim}[layer_{rect}]$$

where $area_{rect}$ is the area of the rectangle, and $perim_{rect}$ is the rectangle's perimeter. The $K_{area}[layer]$ and $K_{perim}[layer]$ are constants for each mask *layer*. The perimeter component of the capacitance tries to account for the sidewall capacitance of diffusion regions and is negligible for other layers. The capacitance layer constants are given values from a technology file that iCHARM reads upon startup.

The capacitance extraction procedure is shown in Procedure 4.4. The procedure uses the CreateGraph() procedure to connect each rectangle with its adjacent rectangles. Then, the capacitance of the net is calculated by summing the contribution of each rectangle. The total perimeter of the net is calculated by adding together the perimeter of each rectangle, but the

---

**Procedure 4.4 - Capacitance extraction procedure**

```
{
  CapExtract()
  Input: net: contains a list of rectangles to calculate capacitance for.
  Output: The net's capacitance has been calculated.
}
CapExtract(net)
begin
  CreateGraph( net's rectlist );
  cap ← 0;
  for all rects, prect, in net's rectlist do begin
        area ← area of prect;
        perim ← perimeter of prect;
        for all edges, e, of prect do          { adjust perimeter to exclude abutting edges }
          perim ← perim - length of common edge of prect and e's rect2;
        cap ← cap + area × K_area[layer_prect] + perim × K_perim[layer_prect];
  end;
  net's capacitance ← cap;
end;
```

---

common edge of abutting rectangles must be deducted from the total perimeter value. For example, the total perimeter of two rectangles is shown emboldened in Figure 4.9. In this situation, the common segment must be deducted from the total perimeter.

### 4.2.3. Resistance extraction

Resistance extraction presents more difficulties than capacitance extraction, since resistance depends on current flow through a conductor. To accurately calculate the resistance of a region between two contacts, the Laplace equation, $\nabla^2\psi = 0$, must be solved with the boundary conditions determined by the region and the contacts. Numerical techniques such as the finite-element method (FEM) may be used to solve for the resistance but this is practical only for a small number of rectangles.

Simple formulas are used in iCHARM to make the resistance extraction of medium-sized layouts practical, but of course without the accuracy of FEM techniques. The iCHARM program uses the resistance extraction procedure in iCPEX [15]. Other references include Horowitz [6].

#### 4.2.3.1. Resistance formulas

The conductor geometries for the resistance formulas in this section are shown in Figure 4.10. For long, straight interconnect lines, the simple one-dimensional resistance formula is used:

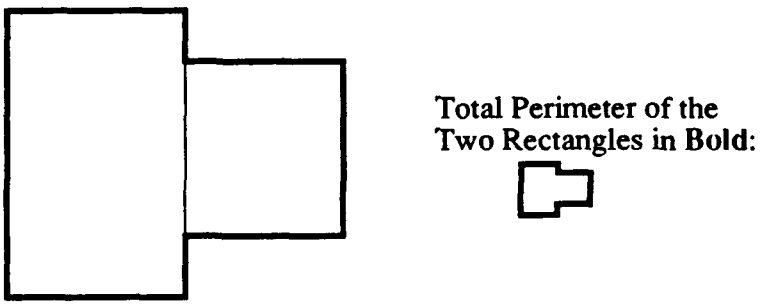
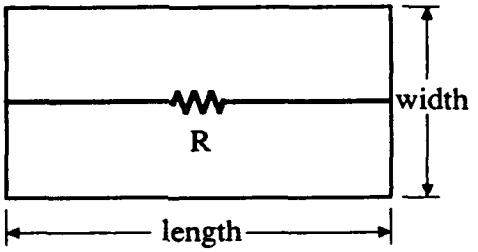



**Figure 4.9 - The perimeter of two rectangles**

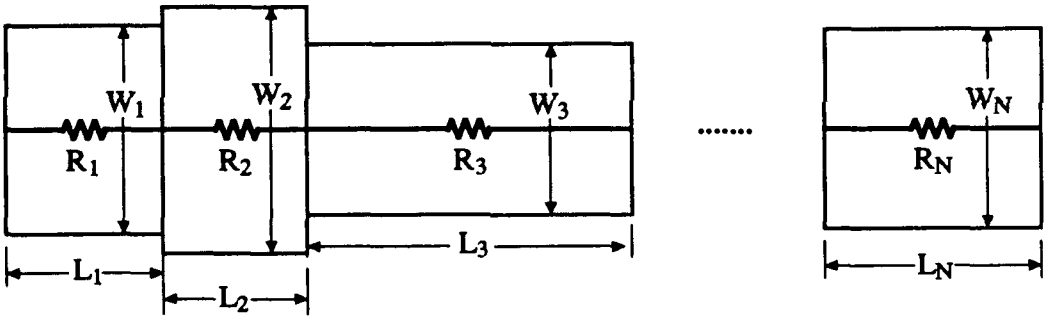
$$R = R_{sh}\frac{length}{width}$$

where $R_{sh}$ is the sheet resistivity of the layer of the rectangle. The $\frac{length}{width}$ factor is the number of square regions between the ends of the rectangle.

**Simple Conductor**




**Abutting Rectangles on the Same Layer**




**Right-angle Bend**

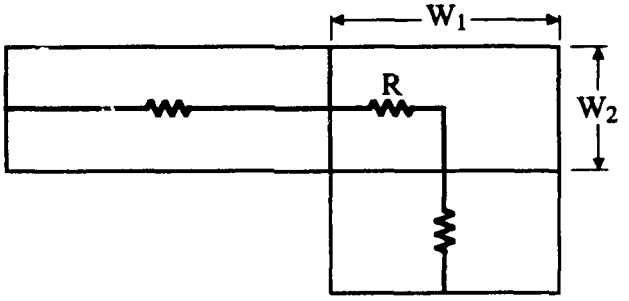



**Figure 4.10 - Resistance formulas**

For abutting rectangles of the same layer (as shown below) with differing widths, the total resistance is calculated by

$$R = R_{sh}\sum_{i}^{N}\frac{L_i}{W_i}$$

This assumes that the direction of current flow is known to be across the abutting rectangles and parallel to the length measurement. In Su [15], the error in this equation is reported to be hard to control but is tolerated in order to keep the calculation simple.

The resistance through a right-angle bend is not as straightforward as counting squares due to current crowding effects. An analytic formula from Su [15] (first given in Hall [5]) is used, which assumes $a \geq 1$:

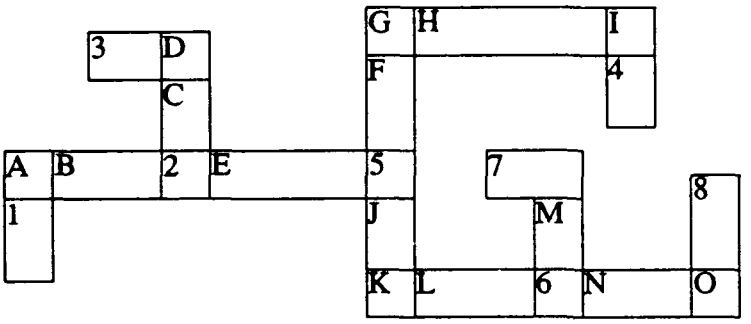
$$R = \left[\frac{1}{a} - \frac{2}{\pi}\ln\left[\frac{4a}{a^2+1}\right] + \left[\frac{a^2-1}{\pi a}\right]\cos^{-1}\left[\frac{a^2-1}{a^2+1}\right]\right] R_{sh}$$

where $a = \frac{w_1}{w_2}$.

#### 4.2.3.2. Resistance extraction process

To handle arbitrarily-shaped conductors in the "real world," the direction of current flow must be estimated. To do this, the rectangles of a net are decomposed into another set of nonoverlapping rectangles that cover the same area but indicate the direction of current flow. All of the rectangles of the net are classified as either *branch* or *knot* rectangles. This step is called branch creation.

A *knot* rectangle is defined to be a rectangle that abuts with more than two rectangles, or with just one other rectangle. A *port* is a rectangle that forms the gate, source, or drain terminal of a transistor. Ports are also considered to be knot rectangles. Finally, all other rectangles are classified as *branch* rectangles. Figure 4.11 shows a set of nonoverlapping rectangles created by rectangle decomposition and identified as either branches or knots. Notice in the figure that the branch rectangles define paths, or *branches*, between each knot for current flow. The resistance formulas given above are used to calculate the resistance for each current path or branch.




**Knots or Ports:** 1,3,4,7,8

**Knots:** 2,5,6

**Branch Rects:** A,B,C,D,E,F,G,H, I,J,K,L,M,N,O

**Branches:** 1,A,B,2 6,M,7
2,C,D,3 6,N,O,8
2,E,5
5,F,G,H,I,4
5,J,K,L,6

**Figure 4.11 - Rectangle decomposition for branch creation**

The actual steps in resistance extraction are the following; the reader may follow along with the example given in Figure 4.12.

(1) The electrical connectivity graph is first created to connect adjacent rectangles of the net.

(2) The graph is used to perform rectangle decomposition and branch creation. First, the rectangles are converted into a set of nonoverlapping rectangles that cover the same area as the original rectangles and are in maximal horizontal form. Then the long horizontal rectangles are combined vertically. In this step, if a rectangle is at least twice as wide as it is tall and it completely abuts with another rectangle above or below, then the two rectangles are combined.

(3) Next, the rectangles are split to create knots. Abutting rectangles that create "T-junctions" are split by projecting the vertical sides of the bottom rectangle across the top rectangle (see Figure 4.13). This is the final form for the rectangles of the net.

(4) The knot and branch rectangles are then identified. Each knot is given a node number because the knot rectangles will become the terminals for the resistances. The transistor terminals also have node numbers since ports are knots.

(5) Once the branch and knot rectangles are identified, each branch is traversed to calculate the resistance of the branch using the simple formulas.

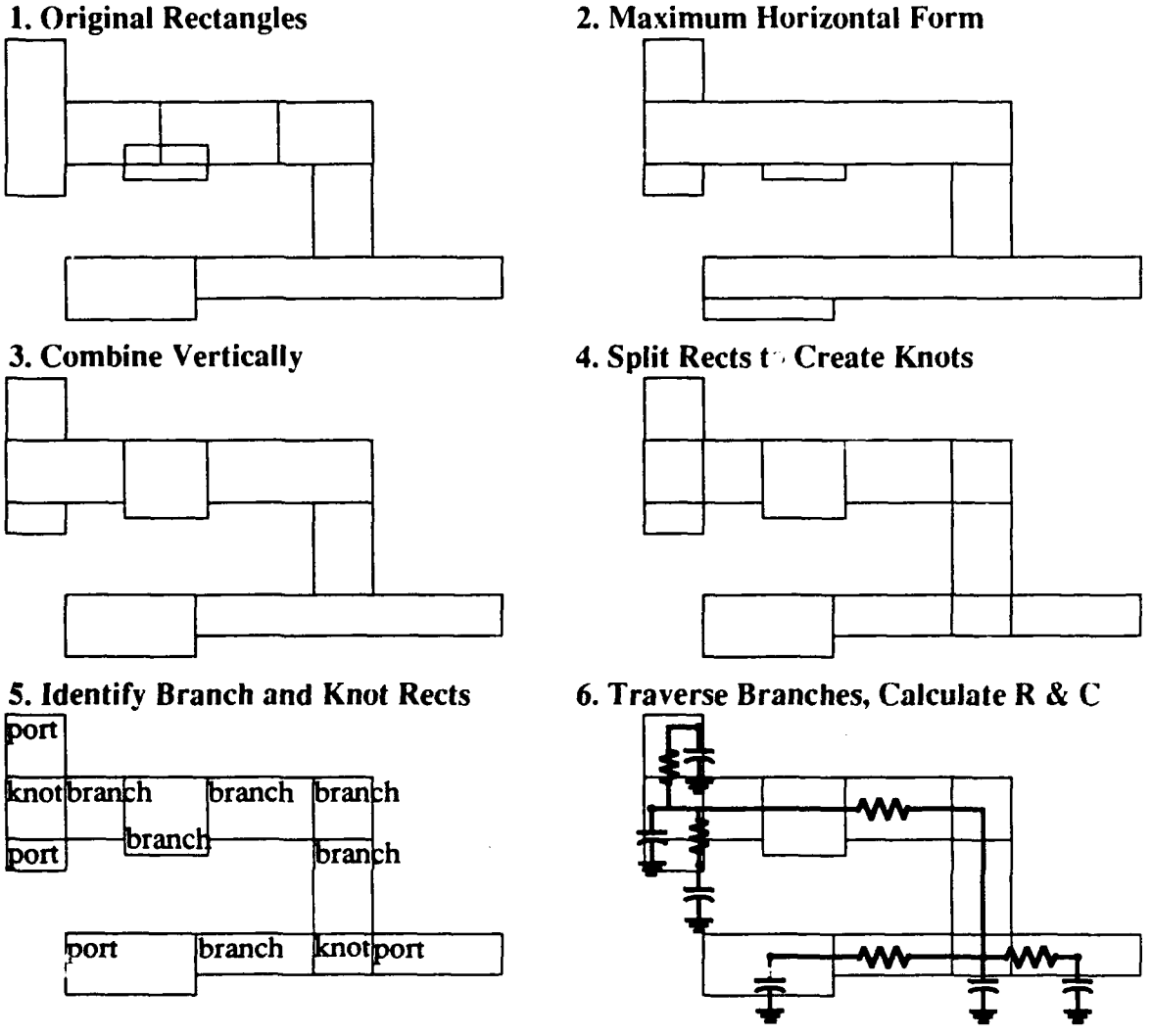



**Figure 4.12 - Resistance extraction**

When the resistance is calculated for each branch, the substrate capacitance is computed as well. The capacitance contribution of each rectangle in the net is calculated as in Procedure 4.4, with the length of any abutting side deducted from the perimeter value of the rectangle. The capacitance of each knot rectangle is calculated first, then the capacitance of each branch rectangle is computed. The branch capacitance is computed when the branches are also being

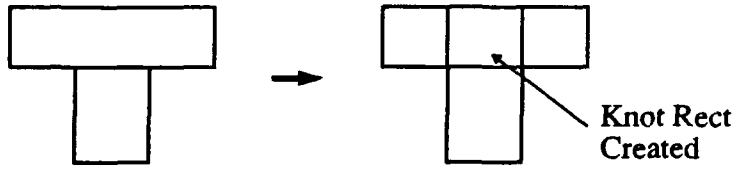



**Figure 4.13 - Splitting rects to create knots**

traversed for the resistance calculation. The capacitance of each branch is lumped onto either end (which is a knot) equally, that is, one-half of each branch capacitance is added to the capacitance of each knot of the branch. The total capacitance of each knot is then reported as the capacitance of the knot's node number with ground.

## 4.3. Power Bus Extraction

In addition to functional and performance verification, circuit extraction can be used to predict the reliability of a given layout. This section describes a reliability analysis system using the CREST, JET, and iCHARM programs. The system uses the extracted output from iCHARM and is shown in Figure 4.14.

### 4.3.1. A reliability analysis system

One failure mechanism is caused by *electromigration* in the metal lines that make up the power supply busses of a chip. Electromigration occurs when the current density is high in sections of the power busses which can, over time, displace the metal atoms in the busses to create gaps. The failure rate of a design can be specified by the median-time-to-failure (MTTF) of the design that is caused by electromigration. The MTTF can be estimated from the current drawn by the circuit from the power supply busses.

Probabilistic simulation has emerged as a promising and computationally-efficient method to estimate the current drawn by a circuit. The CREST program [10] uses such an approach to produce *expected current waveforms* for each point in a circuit where current is drawn from the power busses. Instead of exhaustively simulating the effect of every possible input on the current load of the power supplies, CREST uses probabilistic inputs to a circuit to develop

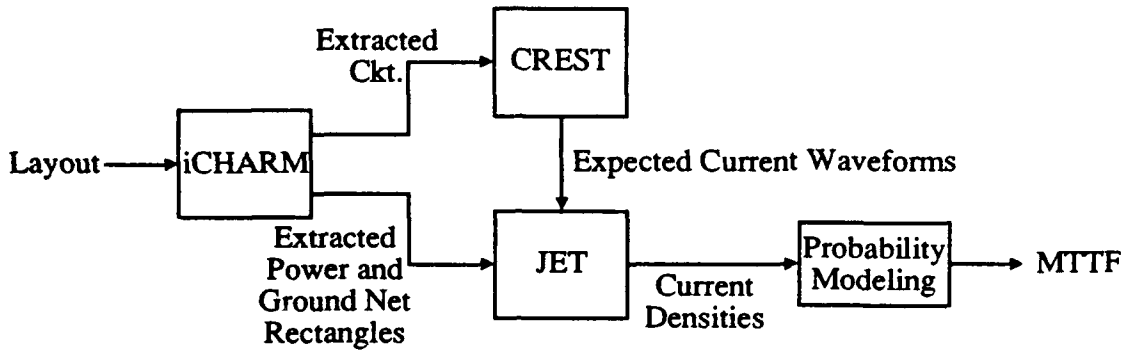



**Figure 4.14 - A reliability analysis system using iCHARM, CREST, and JET**

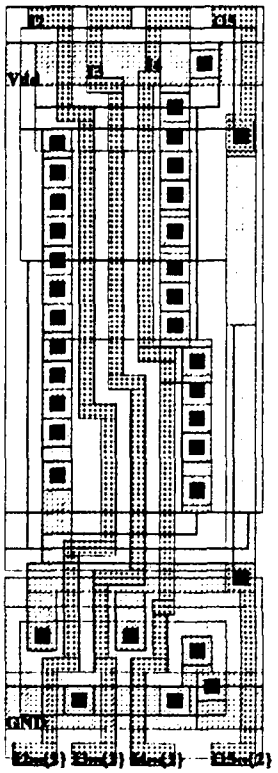
expected current waveforms at every contact point on the power busses. The contact points are located where transistors connect to the power or ground net.

The JET (current density Extraction Tool) program [3] was developed to calculate the current densities within the metal lines of the power busses. The JET program takes as input the rectangles of the power busses and creates an RC network model of the power busses similar to the resistance extraction done by iCHARM. Using the expected current waveforms at the contact points from CREST and the geometry of the power busses, JET produces a current density waveform for each resistive branch in the power busses. From the current densities, MTTF values can be calculated for each section of the power bus.

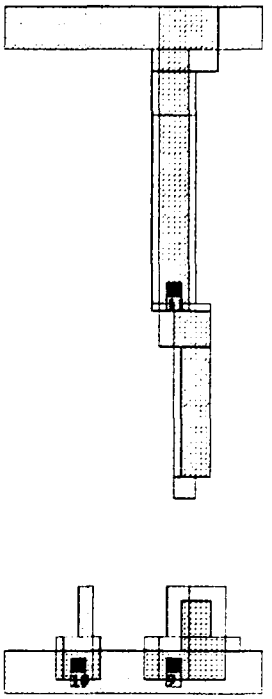
In Figure 4.14, the reliability system being described is shown. The iCHARM program is run in its "power bus extraction" mode to produce the output used by CREST and JET. The extracted circuit description produced by iCHARM is passed on to CREST. The CREST program outputs an expected current waveform at each power bus contact point. The iCHARM program outputs all of the rectangles of the power and ground nets for JET.

A sample of the input and output for the power bus extraction mode is shown in Figure 4.15. The layout shown is a three-input CMOS NOR gate.

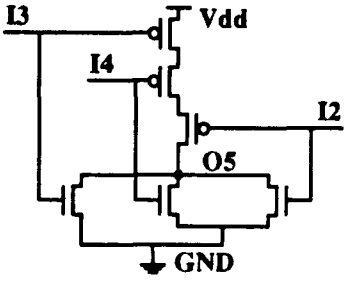
**Layout**




**Extracted Layout with Labeled Contacts**




**Extracted Circuit**




**Extracted Circuit with Split and Renumbered Contact Points**

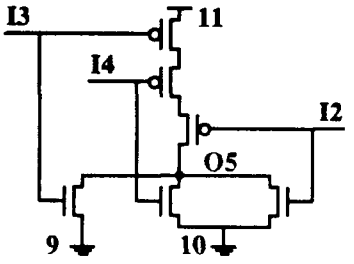



**Figure 4.15 - Inputs/Outputs of iCHARM for power bus extraction**

The power bus extraction mode also splits and renumbers each contact point. Each transistor connection to the power or ground net is given a new net number in the extracted circuit output. In the layout that is sent to JET, the new net number is also used to label the location of a contact window where the transistor connects to the power bus. In the top left of Figure 4.15, the complete layout is shown. The rectangles of the power and ground nets that are given to JET are in the top right of Figure 4.15. The schematic of the regular extracted output of iCHARM is shown in the bottom left of Figure 4.15; the renumbered output given to CREST is shown in the bottom right. In Figure 4.15, there are two renumbered contact points ("9" and "10") where the three transistors connect to the ground bus since two transistors share one connection to ground.

The contact points are renumbered and labeled so that CREST will output current waveforms using the new net number. The JET program is able to input the current waveform to the corresponding transistor contact point in the layout by finding the label with the same net number. Outputting the rectangles for the power and ground nets is straightforward in iCHARM since they are already collected in the geometric extraction step. The major problem is finding the correspondence between a power bus contact point in the layout and the transistor that is connected to it.

### 4.3.2. iCHARM's power bus extraction mode

A scanline algorithm is used to split and renumber the transistor contact points. The power bus extraction procedure, which is presented in Procedure 4.5, is run after the geometric extraction step. The scanline algorithm takes the rectangles of either the power or ground net as input and produces a list of *diffusion island* structures. Each diffusion island structure holds a list of abutting diffusion rects and a list of metal-diffusion contact rects that are connected to them. The diffusion rects that define a diffusion island form a contiguous "island" of rectangles.

Once all of the diffusion islands have been collected, the renumbering is done. One contact rect is output for each diffusion island. It is chosen to be roughly in the middle of the diffusion island. A label with the new net number is output with the same location as the chosen contact rect. The drain/source connections of transistors connected to each diffusion island are renumbered. This involves using the *un.tranpt* pointer of each drain/source diffusion rectangle to reach the connected transistor. The *snetnum* or *dnetnum* field of the transistor is renumbered depending on whether the power bus is connected to the source or drain of the transistor.

## Procedure 4.5 - Power bus extraction procedure

```
{
  ExtractPwrBus()
  Input: net: a power or ground bus net. If rects in net's rectlist are drain/source rects, their
         un.tranpt field has a pointer to the list of transistors that the rect is connected to.
  Output: The net's rectangles have been output and the contacts renumbered.
}
ExtractPwrBus(net)
begin
  ScanlineAlg( net's rectlist );      { create diff-island structure }
  ProcessDiffIslands(net);
end;

{
  AddToScanline()
  Input: prect: the rectangle to process and add to scanline.
  Output: If prect is a diff or contact rect, then it is added to a new or existing diffusion island
          and the scanline; else just output prect to the extracted power bus file.
}
AddToScanline(prect)
begin
  if prect is a diff or contact rect then
  begin
      make a new diffusion island, newdi, and put prect in it;
  { find all rects already in scanline that touch prect }
      for all rects, srect, in scanline that touch prect do
      begin
        if srect and prect connect then
            combine the diffusion islands that contain prect and srect;
      end;
      add prect to scanline;
  end;
  else
      output prect to extracted power bus file;
end;

{
  DeleteFromScanline()
  Output: All rects past curscan are deleted from the scanline structure.
}
DeleteFromScanline()
begin
{ flush all rects in scanline past curscan }
  for all rects, srect, with Xmax < curscan do
      srect ← Pop(scanline);
end;
```

In Figure 4.16 the diffusion island struct that was built for the ground net of Figure 4.15 is shown. There are two diffusion islands built for the ground net. Only one contact rect is extracted for each diffusion island. The new net number labels of "9" and "10" are output at the contact rect locations. The diffusion rects that have connected linkel structs are drain/source

---

**Procedure 4.5 - Power bus extraction procedure (continued)**

```
{
  ProcessDiffIslands()
  Input: net: a power or ground bus net. Upon entry, the diff island structure has been created.
  Output: For each diff island,
      1. A label is output at the location of one contact in the diffusion island using the new net
        number.
      2. The transistor terminals that are connected to drain/source diff rects are renumbered with
        the new net number.
}
ProcessDiffIslands(net)
begin
  newnetnum ← total net count of the circuit + 1;
  for each diffusion island, di, do
  begin
      get the median contact rect, contrect, in the diffusion island;
      output contrect to extracted power bus file;
      output a label of "newnetnum" at contrect's location to the extracted power bus file;
      for all diff rects, drect, of di do
      begin
        output drect to extracted power bus file;
      { renumber the transistor terminals with newnetnum }
        if drect a drain/source rect for a transistor then
        begin
            for all transistors, tran, in drect's linkel list do
            begin
              if snetnum of tran = netnum of net then
                  tran's snetnum ← newnetnum;
              else if dnetnum of tran = netnum of net then
                  tran's dnetnum ← newnetnum;
            end;
        end;
      end;
      increment newnetnum;
  end;
end;
```

---

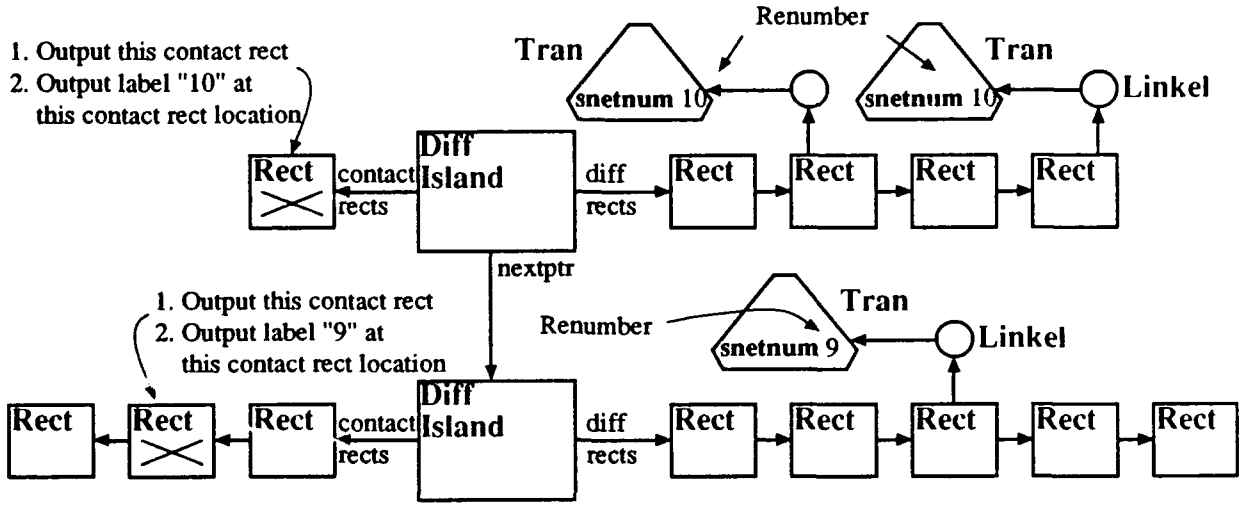



**Figure 4.16 - Diffusion island structure built for the ground net of Figure 4.15**

rects for transistors. The connected transistor's *dnetnum* or *snetnum* field is renumbered using the same net number as the label.

# CHAPTER 5

# iCHARM HIERARCHICAL EXTRACTION

## 5.1. Hierarchical Geometric Extraction

The use of hierarchy to accelerate the extraction process is discussed in this chapter. As mentioned before, hierarchical extraction involves extracting each unique cell in the design hierarchy only once and referring to the previously extracted information whenever an instance of the cell is encountered. The alternative is to flatten the entire design hierarchy which replicates each instance of a repeated cell in memory and makes it necessary to repeat the extraction procedure for each instance. Hierarchical extraction therefore offers a savings in space as well as time. The amount of savings is a function of the amount of regularity, or number of repeated cells, in the design being extracted.

Flat extraction is done on a cell that contains only mask geometries since any hierarchical structure has been flattened away. The flat extraction procedure can be extended into a hierarchical one by enabling the extraction of a cell that contains instances as well as geometries. The difficulty in the technique lies in the fact that a cell can be changed by any geometry that overlaps the cell when it is instantiated. For example, a polysilicon line that is run over an instance can short two connections in the interior of the instance or create a transistor where it crosses a diffusion wire. Overlapping geometries can similarly break connections in an instance or remove an existing transistor. Two instances may themselves overlap which can also affect the internal connections of the instances.

Hierarchical extraction may be separated into techniques where overlapping is allowed or not allowed. If overlapping is disallowed, the design of the extractor is simplified but the designer's freedom is constricted. The extraction process is made easier since cells only interact on their boundaries, and their internal circuitry cannot be changed when they are instantiated. Extraction of cells that contain both instances and geometries (non-leaf cells) is done by creating nodes from the geometries, then connecting up the nodes that are at the instance boundaries.

Clearly, the use of restricted design styles to optimize a CAD tool is not practical. In fact, allowing cell overlap is often desirable since it may reduce area. The existing approaches for hierarchical extraction will now be surveyed. All of these programs create hierarchical data structures to represent the input layout, then extract each unique cell only once while attempting to handle the special cases created by overlapped instances.

### 5.1.1. Existing approaches to hierarchical extraction

The iCPEX program [15] implements a hierarchical extraction capability that handles a minimal amount of overlap. It is meant to be used on designs where overlap occurs only sparingly. The method that iCPEX uses first traverses the hierarchy of the design to detect overlap. When overlap is detected, the overlapped instance is flattened into its parent cell. If overlap still occurs with the subcells of the flattened instance, flattening is continued until no overlap occurs. One can see that in designs where overlap occurs at all levels of the hierarchy, this method will flatten the entire design.

A typical situation that is "hard" for hierarchical methods is shown in Figure 5.1. The cell contains the entire chip, and consists of the padframe cell that "doughnuts" the rest of the

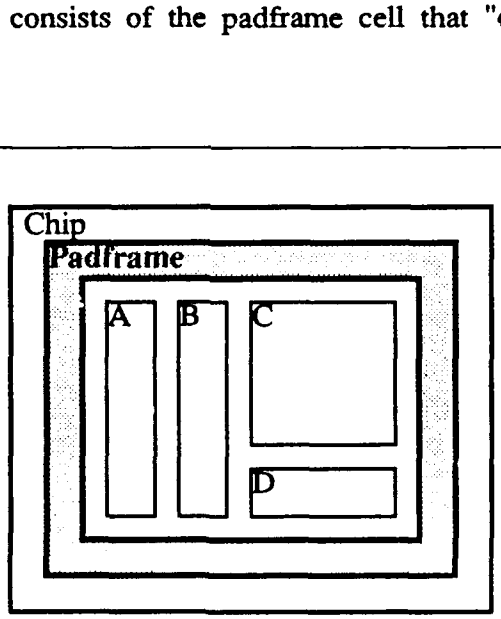



**Figure 5.1 - A difficult case for hierarchical extraction**

functional cells. Since the padframe cell overlaps all of the other cells, the iCPEX method would flatten the entire hierarchy.

The *disjoint transformation* method proposed by Newell [11] handles overlap by recognizing similar overlapped areas and transforming the overlapped design into an equivalent one with no overlap. In Figure 5.2, there are three instances A, B, and C that overlap. The hierarchy is transformed by partitioning the geometries in A and B into A', B', and C'; the geometries in B and C are partitioned into the new cells C', D' and E'. The resulting hierarchy of A', B', C', D', and E' is nonoverlapping at the present level, but may contain further overlap within each cell. The transformation process is carried out top-down until all overlap is removed by transformation. The method also attempts to recognize transformed cells that are identical to reduce the number of cells that are created.

The primary advantage to this technique is that once the transformation has been done, the nonoverlap extraction procedure may be used. However, in some cases the transformed hierarchy may be as large as the completely flattened hierarchy. Furthermore, an additional translation step is required to report the extracted results, since the extraction is done on the transformed

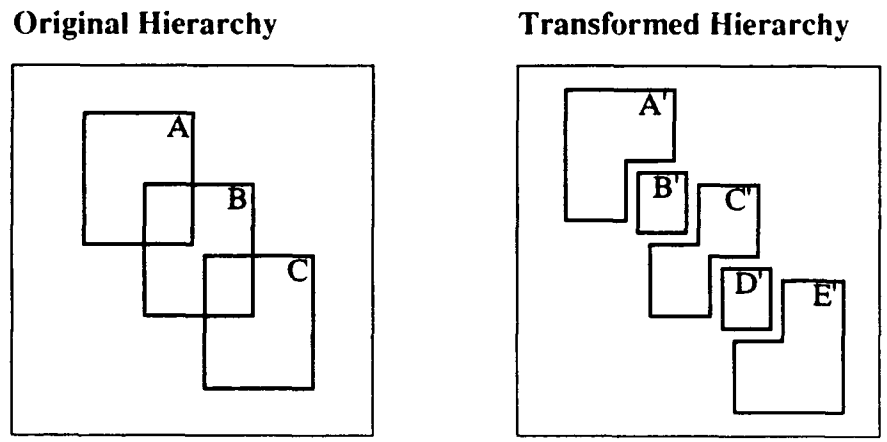



Figure 5.2 - The disjoint transformation technique

hierarchy but the node and transistor labels should be reported to the user in terms of the original hierarchy.

Similar to iCPEX, the Magic circuit extractor [14] handles overlapped instances by selective flattening. However, instead of flattening the entire overlapped instance, only the area of overlap is flattened into the top level cell. In Figure 5.3, only the geometries shown are flattened into the parent cell. This reduces the amount of geometries that are flattened, but Magic will still flatten the entire hierarchy for the padframe example given in Figure 5.1.

### 5.1.2. The iCHARM implementation

The iCHARM implementation of hierarchical extraction used the flat extraction code of the PACE program as a starting point. The goal was to extend the basic extraction module to create a hierarchical extraction capability by making use of existing code, avoiding any duplication of functions, and minimizing the addition of too many new subroutines. In addition, it was felt that

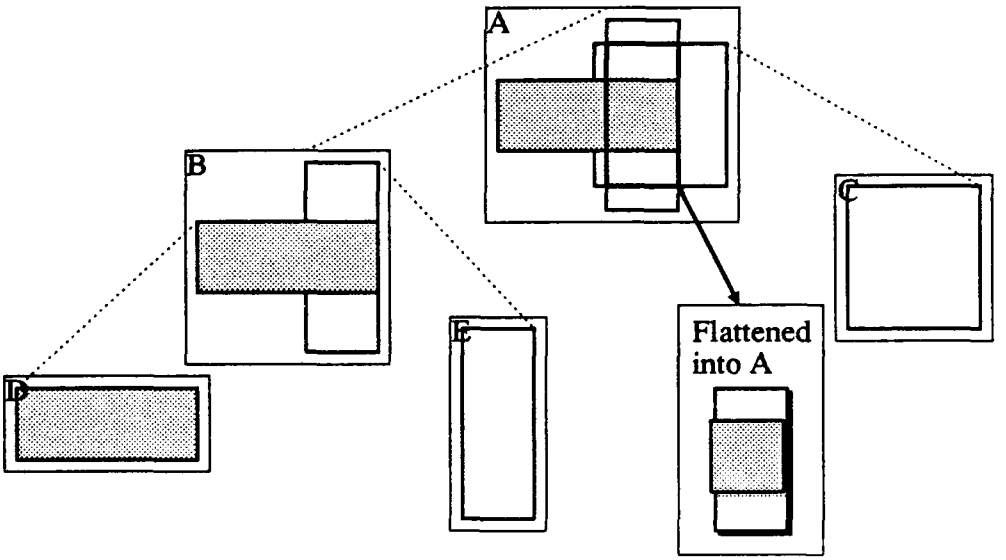



**Figure 5.3 - How Magic handles overlap**

it was important to handle the padframe test-case without flattening because it is a situation that commonly occurs.

Hierarchical extraction was implemented in two stages. The first stage, which will be described first in this section, was implemented under the assumption that the input layout is nonoverlapping. Subsequently, the modifications to allow overlapping were developed.

#### 5.1.2.1. Nonoverlapping hierarchical extraction

In Procedure 5.1, the top-level subroutines for extraction are shown. Hierarchical extraction is done by HierExtract() and flat extraction by FlatExtract(). Both routines call the routine CellExtract(). For hierarchical extraction, the for-loop in CellExtract() processes every cell in bottom-up order. Each subcell of the present cell is checked to see if it has been extracted before the present cell is extracted. If a subcell has not been extracted, CellExtract() is called recursively on it. CellExtract() is called on every cell exactly once. For flat extraction the top-level cell does not have any instances so CellExtract() is executed only on the top cell.

The other routines in Procedure 5.1 — InstantiateTerms() and ReportSubcktHeader() — will be discussed after an introduction to the necessary modifications made to GeometricExtract() to allow it to do hierarchical extraction.

In Figure 5.4, a simple hierarchical extraction example is shown consisting of a layout, its extracted circuitry, and the Spice-format output. Cell A contains a single transistor. Recall that for nonoverlapping designs, all instances connect to the outside world only at their boundaries. These external connections are called the *external terminals* for a cell, and they are created where the geometry of the cell touches the cell boundary. Cell A has four external terminals, but two are connected to the gate of the transistor, so they are part of the same electrical node. In Figure 5.4, the terminals are represented by the thick lines.

Cell B contains an instance of cell A. It has four external terminals as well. Cell B also has four *internal terminals* that are connected to the instance of cell A. The internal terminals are created from the external terminals of cell A when cell A was instantiated. External and internal terminals are analogous to the formal and actual parameters of a subroutine: formal parameters exist on the definition of a subroutine and actual parameters on the call of the routine.

In order to extract a cell that contains geometries and instances, the basic geometric extraction routine must be modified to use the rectangles of the cell and the internal terminals from the instances of the cell. The rectangles of the cell are used to create nets in the usual manner; the

**Procedure 5.1 - Hierarchical and flat extraction procedures that share CellExtract()**

```
{
  HierExtract()
  Input: cell: The top-level hierarchical cell to extract.
  Output: cell and its subcells have been hierarchically extracted.
}
HierExtract(cell)
begin
  CellExtract(cell);
end;

{
  FlatExtract()
  Input: cell: The top-level hierarchical cell to extract.
  Output: cell has been flattened then extracted.
}
FlatExtract(cell)
begin
  if cell is hierarchical then Flatten(cell);
  CellExtract(cell);
end;

{
  CellExtract()
  Input: cell: The cell to extract. The input rects are in cell's rectlist.
  Output: The extracted nets, terms, and transistors for cell in their respective lists.
}
CellExtract(cell)
begin
  for all instances, inst, in cell's instance list do
  begin
  { extract all sub-cells first }
        if inst→defn→done is false then
          CellExtract(inst→defn);

  { instantiate all external terms of inst into cell's internal terminal list }
        InstantiateTerms(cell, inst);
  end;

  cell→done ← true;     { mark cell as done }
  GeometricExtract(cell);
  ParasiticExtract(cell);

  ReportSubcktHeader(cell);   { print .subckt card for hierarchical extraction }
  report transistors and parasitics for cell;
  ReportSubcktEnd(cell);
end;
```

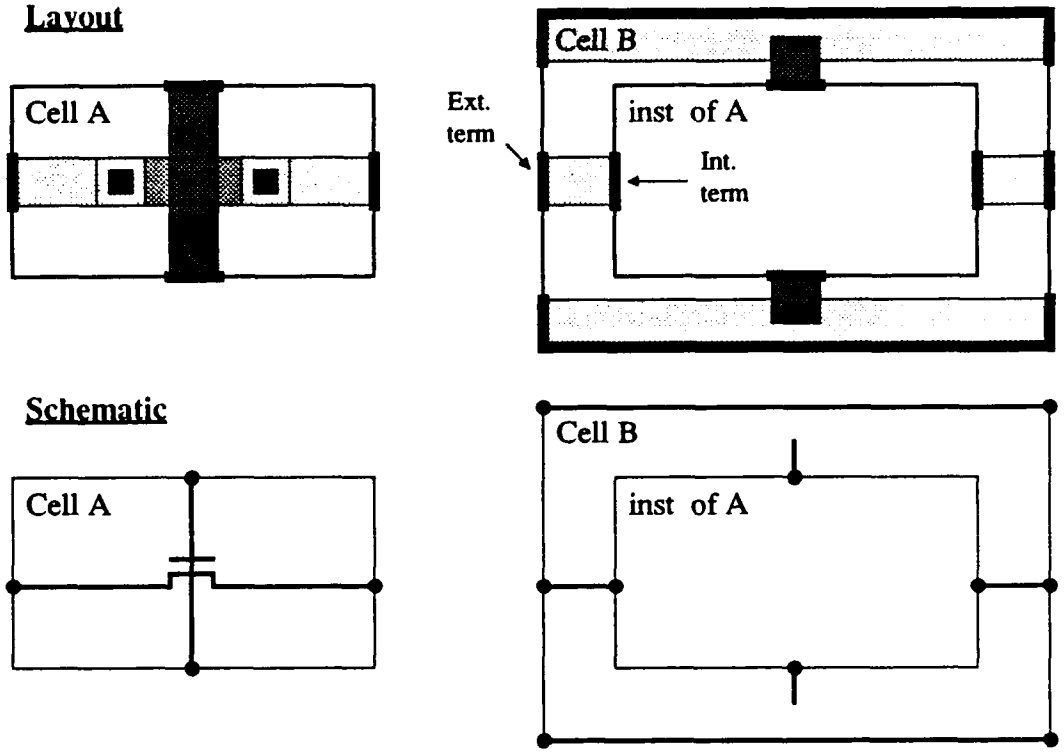



**Figure 5.4 - A simple hierarchical extraction example**

internal terminals are used to connect the nets of the cell to the instances. In addition, the extraction procedure must create external terminals for the cell when rectangles touch the cell boundaries.

The term struct implements both the external and internal terminals. The C-language definition of the term struct is shown in Figure 5.5. The location and layer of a terminal are specified by its *coord* and *layer* fields.

```
/* T E R M - a terminal struct */
typedef struct _term {
   long coord[4];          /* xmin, ymin, xmax, ymax of the terminal */
   char layer;             /* layer number */
   struct _net *setnet;    /* ptr to net (term is in netp→termlist) */
   struct _inst *setinst;  /* ptr to inst (term is in instp→termlist) */
   struct _term *setterm;  /* scratch terminal ptr */
   struct _term *nextnptr; /* next term on the net's termlist */
   struct _term *nextiptr; /* next term on the inst's termlist */
} term, *termptr;
```

**Figure 5.5 - Term data structure**

Figure 5.6 shows how the term struct represents the connectivity of the extracted instances and netsilarly, the terminals of an instance start at the instance's

*termlist* field and follow the *nextiptr* field. This is illustrated in Figure 5.6: internal term T1 connects net N1 and instance I1.

For external terminals, only the connections to nets are needed. The *setnet* field of the term points to its connected net. The list of external terminals for a cell begins at the cell's *exttermlist* field. In Figure 5.6, one external terminal T4 is shown that connects to net N1.

The hierarchical extraction procedure of Procedure 5.1 will now be demonstrated on the example shown in Figure 5.4. In Figure 5.7, the example of Figure 5.4 is again presented along with the data structures that are built for the two cells A and B.

In the absence of instances, geometric extraction is done exactly as before. As the input rectangles are processed, however, a check is made to see if any of the rectangles touch the cell boundary. A touching rectangle creates an external terminal that has the dimensions of the intersection of the rectangle and the cell's bounding box; this is a line segment.

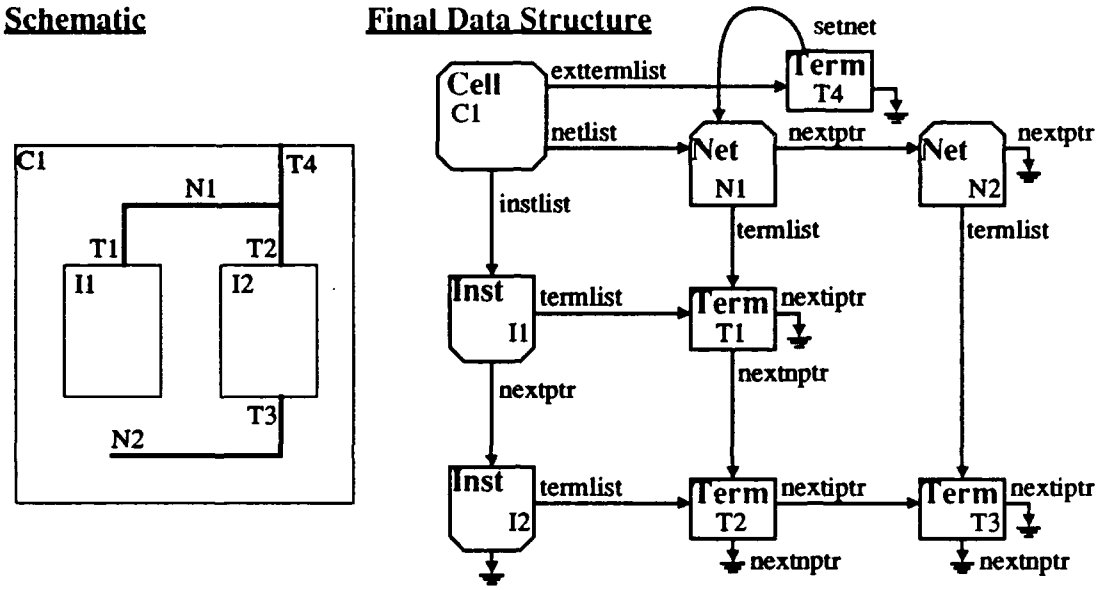



**Figure 5.6 - Cell connectivity represented with term structs**

At the end of the extraction procedure for cell A, four external terminals are created that are connected to three nets. Figure 5.7 shows an important test-case: cell A has a "feed-through" rectangle (the gate of the transistor) that creates two external terminals on either side of the cell. The two terminals are physically separated, so from the outside of cell A it appears that the terminals are not connected. Since it is important to ensure that the two terminals are kept connected, the two external terminals of cell A in Figure 5.7 point to the same net using their *setnet* fields.

After cell A has been extracted, the InstantiateTerms() routine of Procedure 5.1 is called for the instance of cell A in cell B. This routine creates an internal terminal for each external terminal of cells that are instantiated in the present cell. Cell B has an instance of cell A, so four internal terminals are added to cell B's *inttermlist*. When the internal terminals are created, new net structures are also created. To retain the fact that two terminals are connected to the gate net of cell A, the net's *termlist* has two elements as shown in Figure 5.7.

The extraction of cell B is more complicated than the extraction of cell A since cell B contains rectangles and (internal) terminals. The net extraction procedure using scanlines, which

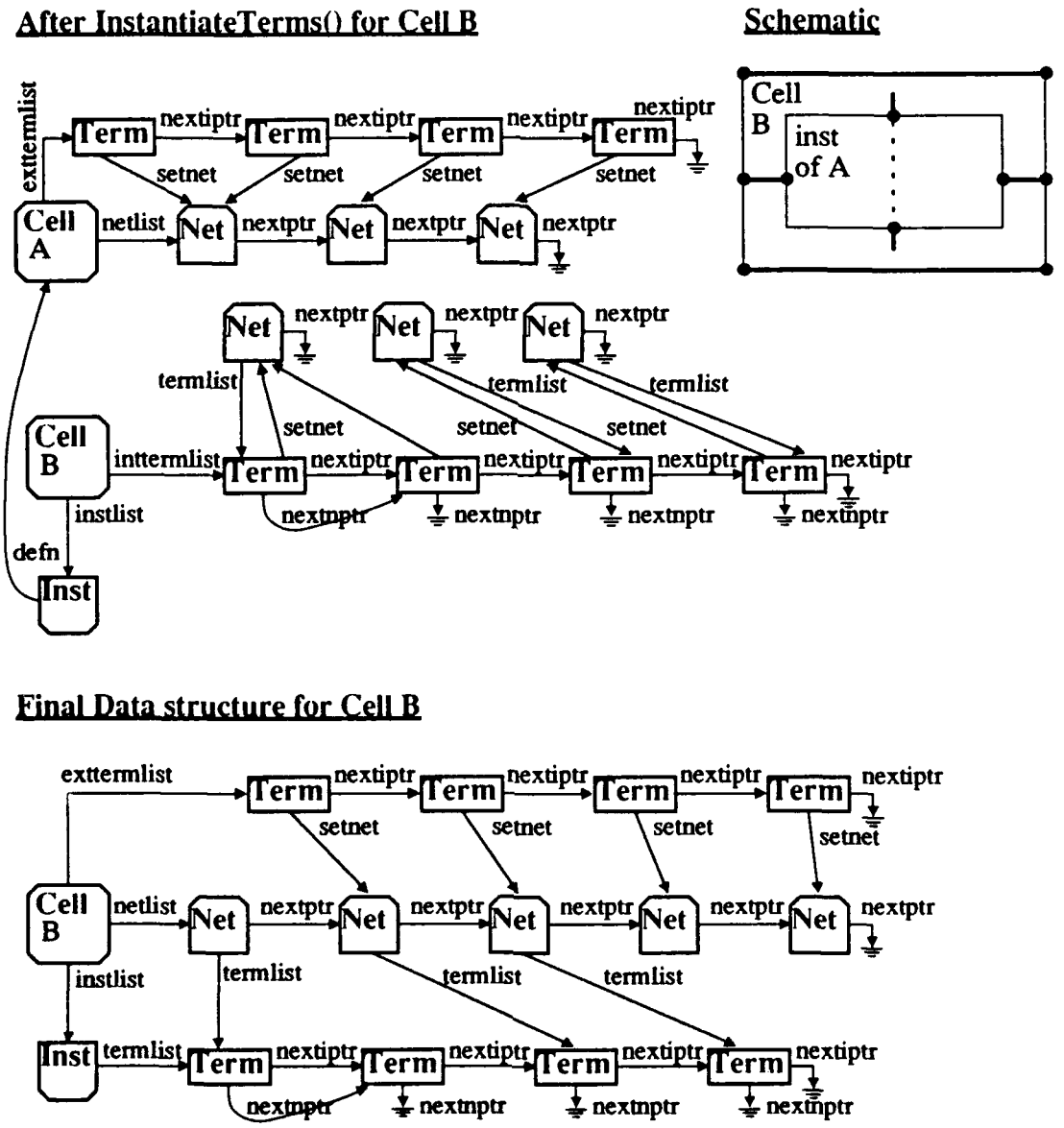



**Figure 5.7 - Hierarchical extraction data structure example**

was shown in Procedure 4.2, must be modified to handle internal terminals. Basically, the AddToScanline() routine must be modified to find and combine touching terminals as well as rects. In addition, when two nets are merged, their *termlists* must also be combined. Again,

external terminals are created when rectangles or internal terminals touch the cell boundary. At the end of the extraction of cell B, the final data structure is shown in Figure 5.7.

Finally, the hierarchical extraction results are reported in Spice format using .SUBCKT statements as shown in Figure 5.4. The .SUBCKT statement reports the extracted results for a cell: the instances that the cell contains, and the *ports*, or externally-connected nets, of a cell. This is done by the ReportSubcktHeader() routine in Procedure 5.1. To report the ports of the cell, the routine visits each terminal in the external terminal list for the cell and prints the *netnum* of the net that is connected to the terminal. More than one external terminal may be connected to a net, but the net is reported only once.

### 5.1.2.2. Handling overlap in hierarchical extraction

It has just been shown that when a nonoverlapping layout is assumed, geometries connect only at the boundary of a cell. When overlap is allowed, the connections to geometries in the interior of a cell need to be recognized. To implement the handling of overlap, a design goal was to recognize the more common cases where overlap causes interior signals to become ports, yet keep the overhead of the method low. The iCHARM program still assumes that most of its cells are not overlapped and most of the connections to a cell occur at its edges.

The most common cases of overlap are shown in Figure 5.8. The cell on the left shows a rectangle overlapping a cell instance and two instances that overlap each other. A more difficult

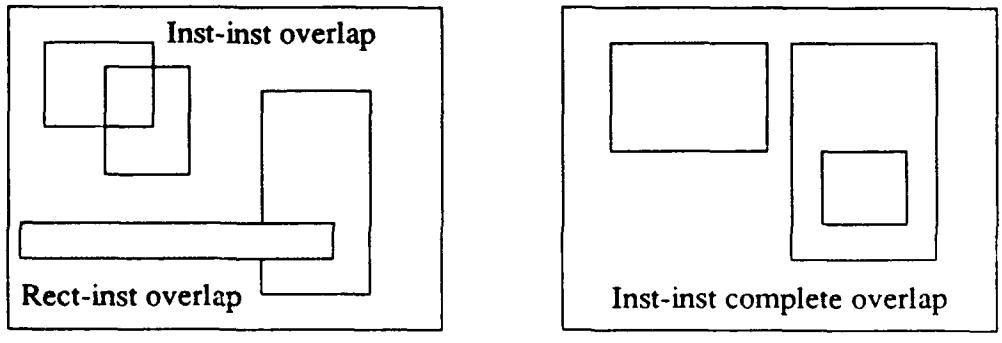



Figure 5.8 - Common overlap situations

case is shown on the right of the figure, where one instance completely envelops another; this is the padframe case that was previously shown in Figure 5.1. An important feature of iCHARM is that the latter case is handled without flattening.

To handle overlap in iCHARM, the basic idea is to visit each unique instantiation of a cell in order to create a list of *overlap-rectangles* for each cell. This is done as a preprocessing step before geometric extraction. The list of overlap-rectangles is a record of all the rectangles that overlap any instance of the cell. For example, in Figure 5.9, three instances of the same cell are shown; as a result, three overlap-rectangles are created for the cell.

Subsequently, when the cell is being extracted (in CellExtract()), every rect of the cell is checked to see if it either

(1) touches the cell boundary, or

(2) touches, and is on a connecting layer, with any overlap-rect.

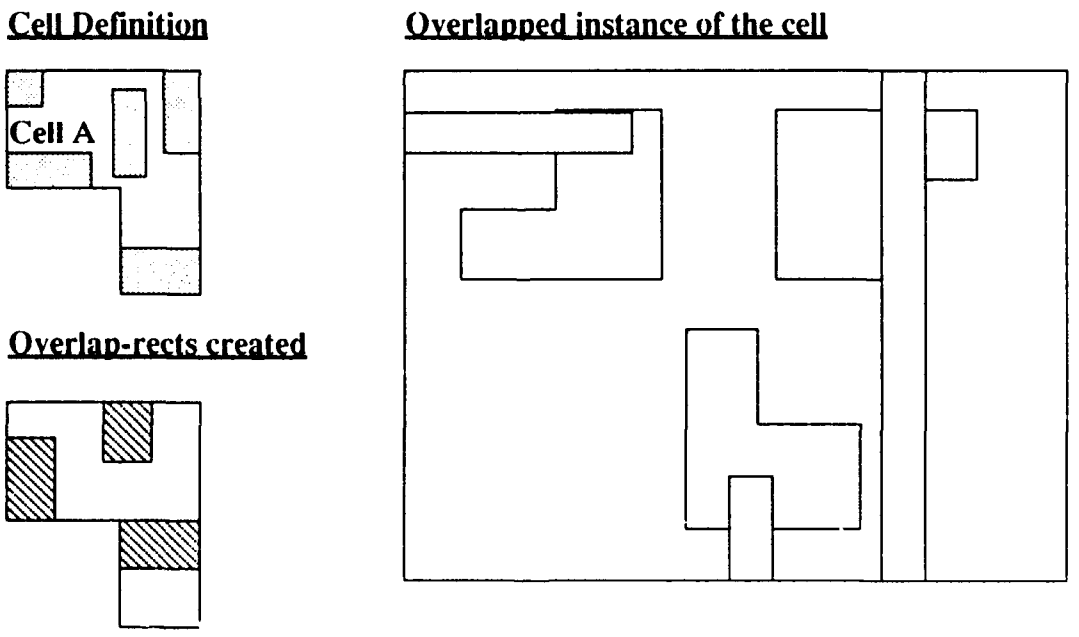



**Figure 5.9 - Overlap rectangles created for three instantiations**

If a rect fulfills either condition, an external terminal is created from the rect. In Figure 5.9, cell A contains five nets, and all five are made into ports since they are all connected to external terminals: four rectangles are on the boundary of the cell, and the internal rectangle touches an overlap-rect.

#### 5.1.2.2.1. Overlap-rect transformation

The overlap-rects for a cell are defined in the cell's coordinate system. The step that creates the overlap-rects finds the intersection of a rect that overlaps an instance of the cell. To convert the rectangle of the intersection into an overlap-rect for the cell, it has to be mapped from the coordinate system of the instance back to the coordinate system of the cell definition. In Figure 5.9, the three instances of the cell are in different orientations relative to the cell definition, thus the rectangles that overlap the instances must be rotated back in terms of the cell definition to create the overlap-rects.

The transform matrix is defined for each instance, therefore, it seems that it also can be used to go from an instance "backward" to the cell coordinate system. Using transformation matrix T to transform a coordinate (x, y) in the cell coordinate system to a coordinate (x′, y′) in the instance coordinate system, the equation previously shown is

$$[x' \; y' \; 1] = [x \; y \; 1]T$$

By multiplying both sides by $T^{-1}$, the inverse of the transformation matrix, one arrives at

$$[x' \; y' \; 1]T^{-1} = [x \; y \; 1]$$

The routine InvTransformCoords() shown in Procedure 5.2 accomplishes the coordinate transformation. The *rect* argument for InvTransformCoords() is the region of intersection between an instance and any rect that overlaps it. The *rect* is translated to the cell's coordinate system to create an overlap-rect for the cell.

In general, all transformation matrices are not invertible. However, in iCHARM the Inverse() routine in Procedure 5.2 is defined only for a restricted class of transformation matrices that occur with 90° rotations:

**Procedure 5.2 - Transforming from the instance coordinate system to the cell definition coordinate system**

```
{
  InvTransformCoords()
  Input: tmatrix: transform matrix of instance
       rect: rect in instance coordinate system
  Output: returns rect transformed into the cell definition coordinate system.
}
InvTransformCoords(tmatrix, rect)
begin
  invtmatrix ← Inverse(tmatrix);
  transform coords of rect using transform matrix invtmatrix;
  return(rect);
end;
```

$$T = \begin{matrix} a/c & b/c & 0 \\ -b/c & a/c & 0 \\ 0 & 0 & 1 \end{matrix}$$

where $c = \sqrt{a^2 + b^2}$. Since only 90° rotations are allowed in iCHARM, $a = \pm 1$ and $b = 0$, or $a = 0$ and $b = \pm 1$; $c = 1$.

### 5.1.2.2.2. The overlap-rect procedure

The preprocessing step to create the overlap-rectlist for each cell will now be presented. The procedure is shown in three parts, in Procedure 5.3 a, b, and c.

In Procedure 5.3a, the modified HierExtract() is shown that calls ProcessOverlap() before CellExtract() to create the overlap-rects. ProcessOverlap() has two major functions. First, it analyzes the rectangles and instances in the given cell and identifies any overlapping areas. Rectangles may overlap instances, or instances may overlap each other. The ScanOverlap() routine uses a scanline algorithm to detect the overlapping rectangles or instances and, if overlap is found, then an overlap-rect is created for the cell of the instance.

ProcessOverlap() is called once by HierExtract() to find the overlapping areas for the entire design hierarchy. Therefore, the second function of ProcessOverlap() is that it traverses the hierarchy such that if a cell is instantiated $n$ times, ProcessOverlap() may be called $n$ times to

**Procedure 5.3a - Hierarchical extraction procedure that handles overlap - part 1**

```
{
  HierExtract()
  Input: cell: The top-level hierarchical cell to extract.
  Output: cell and its subcells have been hierarchically extracted.
}
HierExtract(cell)
begin
  ProcessOverlap(cell);
  CellExtract(cell);
end;

{
  ProcessOverlap()
  Input: cell: The top-level cell to preprocess.
  Output: The overlap-rectlists for all cells have been created.
}
ProcessOverlap(cell)
begin
  if cell→done is false    { if not already done, mark cell as done }
      cell→done ← true;

  PropagateOvlprectsToInstances(cell);
  ScanOverlap(cell);

  for all instances, inst, in cell's instance list do
  begin
      if inst→defn→done is false then
        ProcessOverlap(inst→defn);
  end;
end;
```

find every situation where overlap occurs. For instance, if a cell is instantiated in three different ways, as was shown in Figure 5.9, the cell may be overlapped in up to three different ways as well. Since most designs use repeated instances, ProcessOverlap() limits the number of times that it is called for the same cell. The hierarchy traversal is accomplished in ProcessOverlap() by using the cell's *done* flag, the PropagateOvlprectsToInstances() routine, and by calling itself recursively at the end of the routine.

The ScanOverlap() routine, which is shown in Procedure 5.3b, will be discussed first, and will be followed by a discussion of the hierarchy traversal function of ProcessOverlap().

## Procedure 5.3b - Hierarchical extraction procedure that handles overlap - part 2

```
{
  ScanOverlap()
  Input: cell: The rect-inst and inst-inst overlap will be found for this cell.
  Output: The ovlprectlist for the definitions of the overlapped instances has been updated.
}
ScanOverlap(cell)
begin
  for all instances, inst, in cell's instance list do begin
        let tmprect ← bounding box of inst;
        mark tmprect as an "instance rect."
        add tmprect to cell→rectlist;
  end;
  ScanlineAlg(cell→rectlist);
end;

{
  AddToScanline()
  Input: prect: the rectangle to process and add to scanline.
  Output: Overlap of an instance is found, the appropriate ovlprectlist is updated,
          and prect is added to the scanline structure.
}
AddToScanline(prect)
begin
{ find all rects already in scanline that touch prect }
  for all rects, srect, in scanline that touch prect do
  begin
        if srect and prect overlap then begin
          xsect ← overlap_region(prect, srect);
          if srect and prect are both marked as "instance rects" then begin
                pinst ← instance for prect ;
                sinst ← instance for srect ;
                ProcessOvlpInstances(pinst, xsect);
                ProcessOvlpInstances(sinst, xsect);
          end;
          else if srect or prect are marked as "instance rects" then begin
                if prect is marked as an "instance rect" then
                  inst ← instance for prect ;
                else if srect is marked as an "instance rect" then
                  inst ← instance for srect ;
                tmprect ← InvTransformCoords(inst→tfmatrix, xsect);
                InsertOvlprect(tmprect, inst→defn);
          end;
        end;
  end;
  add prect to scanline;
end;
```

To detect overlapped instances, ScanOverlap() creates special rectangles from the bounding box of the instances and marks them as "instance rects." A scanline procedure is then used to find any rectangle that overlaps an "instance rect." The AddToScanline() routine in Procedure 5.3b is used to detect the overlap. If a rect-inst overlap is found, then the region of overlap is found. The overlap region is found relative to an instance, so it is transformed back in terms of the cell definition of the instance by InvTransformCoords(). The InsertOvlprect() routine is used to add the rect to the definition's overlap-rectlist.

When an instance-instance overlap is detected by AddToScanline(), the ProcessOvlpInstances() routine is called. When an instance is overlapped with another instance, an overlap-rect is made from the area of intersection, and the layer is set to the special *all-connect* layer. This layer is not a real mask layer, but the layer is made to connect to all other layers. If an overlap-rect is on the all-connect layer, then during geometric extraction, an external terminal is created for any rectangle, regardless of its layer, that touches the overlap-rect.

When two instances overlap, all nets in the area of overlap must be "ported," or connected to the outside of the cell. Figure 5.10 shows a situation where this is the case. The instances of cell A and B overlap. Cell A has instances of cell C and D. Cell B contains rectangle B1 that overlaps rect C1 of cell C and rect D1 of cell D since B overlaps A. All rectangles in the overlap

---

**Procedure 5.3b - Hierarchical extraction procedure that handles overlap - part 2 (continued)**

```
{
  ProcessOvlpInstances()
  Input: inst: instance that is overlapped
      xsect: the overlapped region
  Output: An overlap rect is added to the definition of inst.
}
ProcessOvlpInstances(inst, xsect)
begin
  tmprect→layer ← ALL_CONNECT;
  tmprect ← InvTransformCoords(inst→tfmatrix, xsect);
  if tmprect and inst's bounding box are not equal then
       InsertOvlprect(tmprect, inst→defn);
end;
```

---

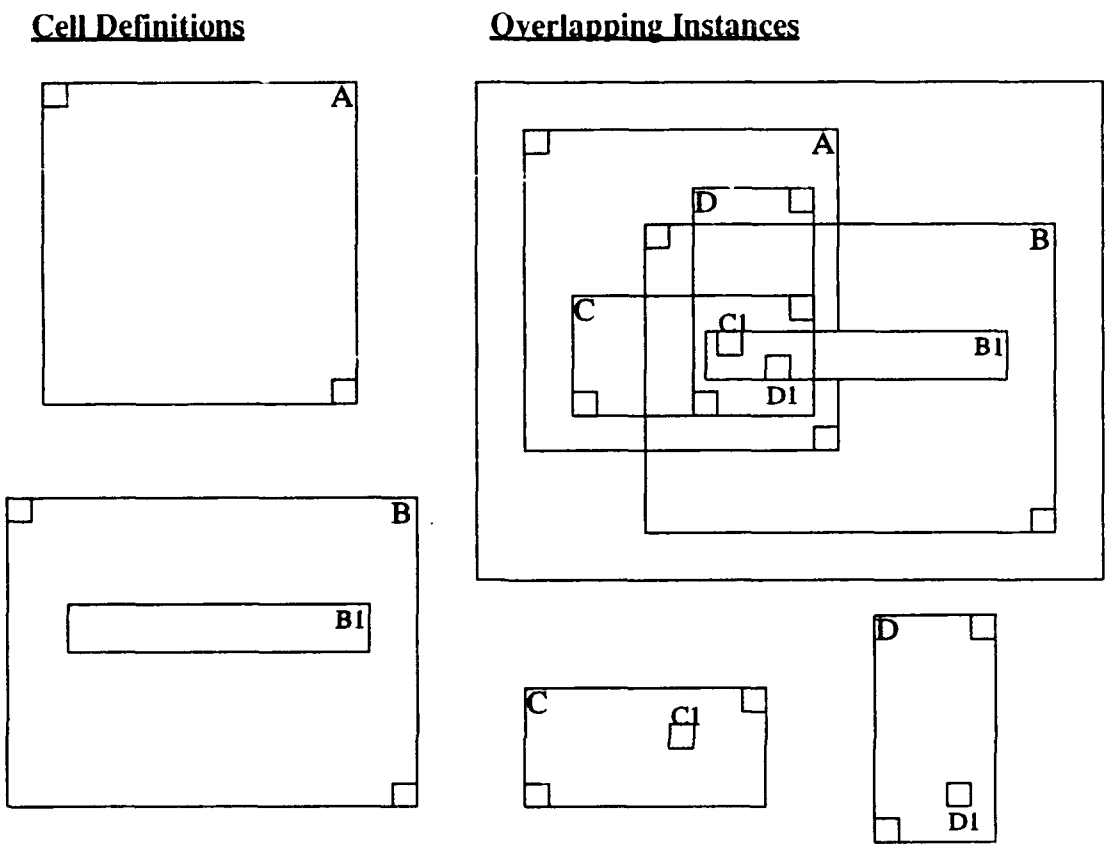



**Figure 5.10 - Instance-instance overlap**

area of A and B must be made into external terminals. This is accomplished by creating an overlap-rect on the all-connect layer, since this causes an external terminal to be made from rects C1 and D1.

ProcessOvlpInstances() contains an important assumption that is made in iCHARM. If an instance β completely overlaps an instance α, then it is assumed that β contains *no rectangles* in the area of α. This is true in the padframe case. In iCHARM, *no* overlap-rect is made that covers the area of the overlapped instance α. Therefore, if β has any rectangles over α, the interior nets of α that are under the rectangle will not be connected to β.

On the other hand, if an overlap-rect to cover α were made, then all of the rectangles of α would be made into external terminals. Furthermore, all of the rectangles of the instances of α — in fact for all of the descendants of α — would become terminals as well. This would create too many external terminals and would slow down the geometric extraction step. The extracted results would not be incorrect, but all internal nets would be made into ports for the cell. Therefore, in iCHARM, if an instance is completely overlapped, then it is assumed that all connections to it will occur at the boundary of the cell.

The routines sketched in Procedure 5.3c control the traversal of the hierarchy by ProcessOverlap(). In ProcessOverlap(), the cell's *done* flag is used to mark if a cell has previously been visited. A cell's flag is immediately set to "done" upon entry to ProcessOverlap(). Then, the overlap-rects for the cell are propagated to the cell's instances by the PropagateOvlprectsToInstances() routine. In Figure 5.10, the overlap of instance A with instance B creates an overlap-rect. The overlap-rect covers more than half the area of instances C and D. In PropagateOvlprectsToInstances(), the intersections of the overlap-rect of A with instances C and D are calculated; then an overlap-rect is created for definition of both instances. The overlap-rect for C in turn causes rect C1 to be made into an external terminal and similarly for D and D1.

ScanOverlap() and PropagateOvlprectsToInstances() may both add overlap-rects to a cell. If an overlap-rect is added to a cell's list, then the cell must be revisited by ProcessOverlap(), since the overlap-rect may have to be propagated to the instances of the cell. Therefore, in InsertOvlprect(), the cell's *done* flag is set to "un-done." Subsequently, at the end of ProcessOverlap(), each instance is checked to see if it needs to be revisited, since its flag may have been unset during the course of the current call. This method visits each cell the minimum number of times and avoids calling ProcessOverlap() for every instance in the design as long as some instances are identical.

## 5.2. Hierarchical Parasitic Extraction

The flat parasitic extraction module in iCHARM as previously described takes the rectangles of the cell as input. The extraction procedure needs to be modified when each cell also contains instances.

**Procedure 5.3c - Hierarchical extraction procedure that handles overlap - part 3**

```
{
  PropagateOvlprectsToInstances()
  Input: cell: The current cell being analyzed.
  Output: If an overlap-rect for cell overlaps an instance, the overlap-rect is propagated
          to the definition of the instance.
}
PropagateOvlprectsToInstances(cell)
begin
{ propagate overlap rects of cell that also overlap the instance }
  for all instances, inst, in cell's instance list do
  begin
        for all rects, ovlprect, in cell's overlap-rectlist do
        begin
          if ovlprect and inst's bounding box overlap then
          begin
                xsect ← overlap_region(ovlprect, inst's bounding box);
                tmprect ← InvTransformCoords(inst→tfmatrix, xsect);
                InsertOvlprect(tmprect, inst→defn);
          end;
        end;
  end;
end;

{
  InsertOvlprect()
  Input: rect: overlap-rect to insert
       cell: cell whose instance is overlapped
  Output: rect is inserted into cell→ovlprectlist, done flag set to reprocess.
}
InsertOvlprect(rect, cell)
begin
  insert rect into cell→ovlprectlist;
{ since ovlprectlist has changed, ProcessOverlap() must be called again }
  if cell→done is true then
        cell→done ← false;
end;
```

### 5.2.1. Hierarchical capacitance extraction

When a cell is extracted hierarchically, capacitance adjustments have to be made for the nets that are connected to instances. Recall that the substrate capacitance equation uses the area and perimeter of each rectangle of a net to compute the net's capacitance. In the flat extraction

procedure, when two rectangles abut, the shared segment must be deducted from the total perimeter of the net. Similarly, when the terminals of two instances abut, or when a rectangle abuts with an internal terminal, the length of the abutment must be deducted from the perimeter of the net. In addition, if a terminal of an instance is overlapped by a rectangle, then the area of intersection must be deducted from the area of the equation. Figure 5.11 illustrates these situations. In Figure 5.11, two rectangles are inside of the instance; the other two rectangles are outside of the instance and connect to the instance by abutment and overlap.

The hierarchical capacitance extraction procedure, which is shown in Procedure 5.4, was created by modifying Procedure 4.4 to handle instances. The scanline procedure of Procedure 4.4 finds the abutting rectangles of a net to adjust the capacitance. Procedure 5.4 must also find the abutting terminals of the net in order to handle instances. To do this in the scanline procedure, rect structs are made with the same coordinates as the terminals and added to the *rectlist*

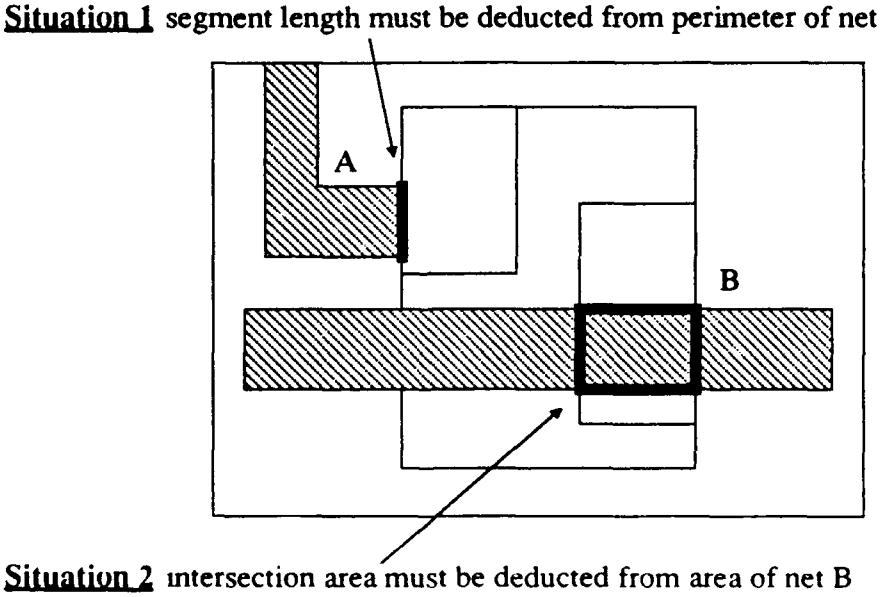



**Figure 5.11 - Capacitance adjustment when connecting to an instance**

of the net. Then the scanline procedure finds the touching rects of the net (including the terminals) by processing the *rectlist*.

Recall that external terminals are created in two ways:

Case 1

A line segment is created for the external terminal when a rect touches the cell boundary. The terminal is the intersection of the rect and the cell boundary.

---

**Procedure 5.4 - Hierarchical capacitance extraction procedure**

```
{
  CapExtract()
  Input: net: contains a list of rectangles and (internal) terminals to calculate the capacitance for.
  Output: The net's capacitance has been calculated.
}
CapExtract(net)
begin
  add the terminals in net's termlist to net's rectlist;
  CreateGraph( net's rectlist );

  cap ← 0;
  for all rects, prect, in net's rectlist do
  begin
      area ← area of prect;
      perim ← perimeter of prect;
      for all edges, e, of prect do
      begin
        if prect and e→rect2 touch and connect then
        begin
            xsect ← area of intersection of prect and e→rect2;
        { adjust area to exclude ½ of overlapping area }
            area ← (area + (area of xsect)/2)
        { adjust perimeter to exclude abutting edges or ½ of perimeter of rect-term xsection }
            perim ← perim - (length of xsect + width of xsect);
        end;
      end;
      cap ← cap + area × K_area[layer_prect] + perim × K_perim[layer_prect];
  end;
  net's capacitance ← cap;
end;
```

---

Case 2

The rect is made into a terminal when the rect touches an overlap-rect (and does not touch the cell boundary).

The external terminal created by Case 1 corresponds to the emboldened area in Situation 1 in Figure 5.11. For Case 2, the emboldened area in Situation 2 of Figure 5.11 is the overlap-rect that is created; an external terminal is created that has the coordinates of rectangle B.

Procedure 5.4 creates an electrical-connectivity graph to connect adjacent rects. The area and perimeter adjustments are made using the graph. Since adjacent rects create two edges in the graph in both directions, one-half of the area adjustment is made for each edge in the graph. For Situation 2 of Figure 5.11, two graph edges are created between the overlapping rect and the terminal. The perimeter adjustment in Procedure 5.4 also works for connections with terminals. The abutment length of Situation 1 in Figure 5.11 is deducted from the perimeter (the emboldened area), and for Situation 2, the perimeter of the terminal (again, the emboldened area) is deducted.

### 5.2.2. Hierarchical resistance extraction

The present implementation of iCHARM does not extract resistances hierarchically. In iCHARM, substantial changes to the data structures and extraction procedures would have to be done in order to implement it. Recall that resistance extraction reports the signals of the design in terms of node numbers, not net numbers. In other words, the *netnum* of the net struct is not used, and the branchnode struct is used to identify the nodes. A branchnode is attached to a rectangle, and rectangles exist in a net struct's *rectlist*.

At present, internal terminals are connected to net structs and instances. External terminals are connected to net structs. To do resistance extraction in iCHARM, the external terminal structs would have to be connected to each rect struct at the boundary of the cell, and internal terminals would have to be connected to each rect that is connected to an instance. An example of this is shown in Figure 5.12. This would be a major change to the program.

## 5.3. Hierarchical Power Bus Extraction

The techniques to do power bus extraction on a flat layout can also be extended to work on a hierarchical layout. The method implemented in iCHARM will be sketched out here.

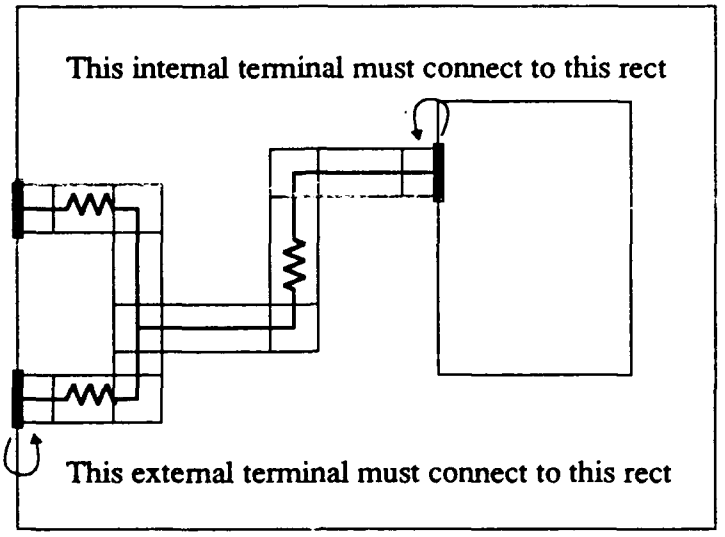



**Figure 5.12 - Terminal structs must be attached to rectangles for resistance extraction**

First, every cell in the hierarchy is geometrically extracted. This creates the connections in the data structures that connect the drain and source rects with their respective transistors.

Next, a procedure is run to mark the nets in every cell that are the power and ground nets. The procedure is started on the top-level cell where the power and ground nets are first identified by name. The internal terminals that are connected to the power / ground nets are found; tracing in the data structures further, the corresponding external terminal of the instance's definition can be found, which is connected to the definition cell's net. The power and ground net for the next lower level is marked in this way. The procedure is then recursively called on every cell. The cell's done flag is used in a way similar to ProcessOverlap() to limit the number of cells that are visited.

Once all of the power and ground nets are marked, the next procedure is executed in a bottom-up order, i.e., as in CellExtract(), to do the actual power bus extraction. For each cell, ExtractPwrBus() (in Procedure 4.5) is called on each marked net. In ExtractPwrBus() the

rectangles of the extracted power bus are printed out. In the current implementation of iCHARM, the rectangles are printed in CIF format. In addition to this, the instantiation, or the "cell call" (the C statement in CIF), of any instances that are connected to the power / ground net must be output. This involves reproducing the "cell call" statement of the instance using the instance's transform matrix. The instances must be included in the output so that the layout description of the extracted power busses is hierarchical.

# CHAPTER 6

# RESULTS AND CONCLUSIONS

## 6.1. Test Results

The final implementation of iCHARM consists of some 11,000 lines of C code,[1] spread over 18 source files and 4 header files, and runs under the UNIX[2] operating system. The input front-end reads layout descriptions in either Oct or CIF format. During its development, iCHARM was fortunate to have a full-chip layout available to provide test-cases.

### 6.1.1. The 2uchp test chip

The 2uchp design is a 10,000 transistor CMOS layout that was designed at the Rome Air Development Center of the U.S. Air Force. It is so named because the top-most cell is named "2uchp." The design was originally described in CIF, but it was converted to an equivalent Oct description to test the Oct input module of iCHARM.

The 2uchp chip allowed iCHARM to be tested with realistic layout situations. The chip consists of 74 different cells. The design hierarchy of the chip is shown in Figure 6.1. The major cells of the chip are a ram (*ram1*), rom (*rom2*), alu (*alun2*), miscellaneous logic (*PRNG*, *chp_qs*, *pr_leak*), and the padframe (*2ufrm*). Although most cells also contain mask geometries, in Figure 6.1 only the subcells are shown for each cell.

Since the majority of the transistors are in the *ram1* cell, it is the most difficult cell to extract flat. In the *ram1* cell there are two instances of the *mem_blk* cell, which is a 512-bit static ram array.

---

[1] This includes comment lines.

[2] UNIX is a trademark of AT&T Corp.

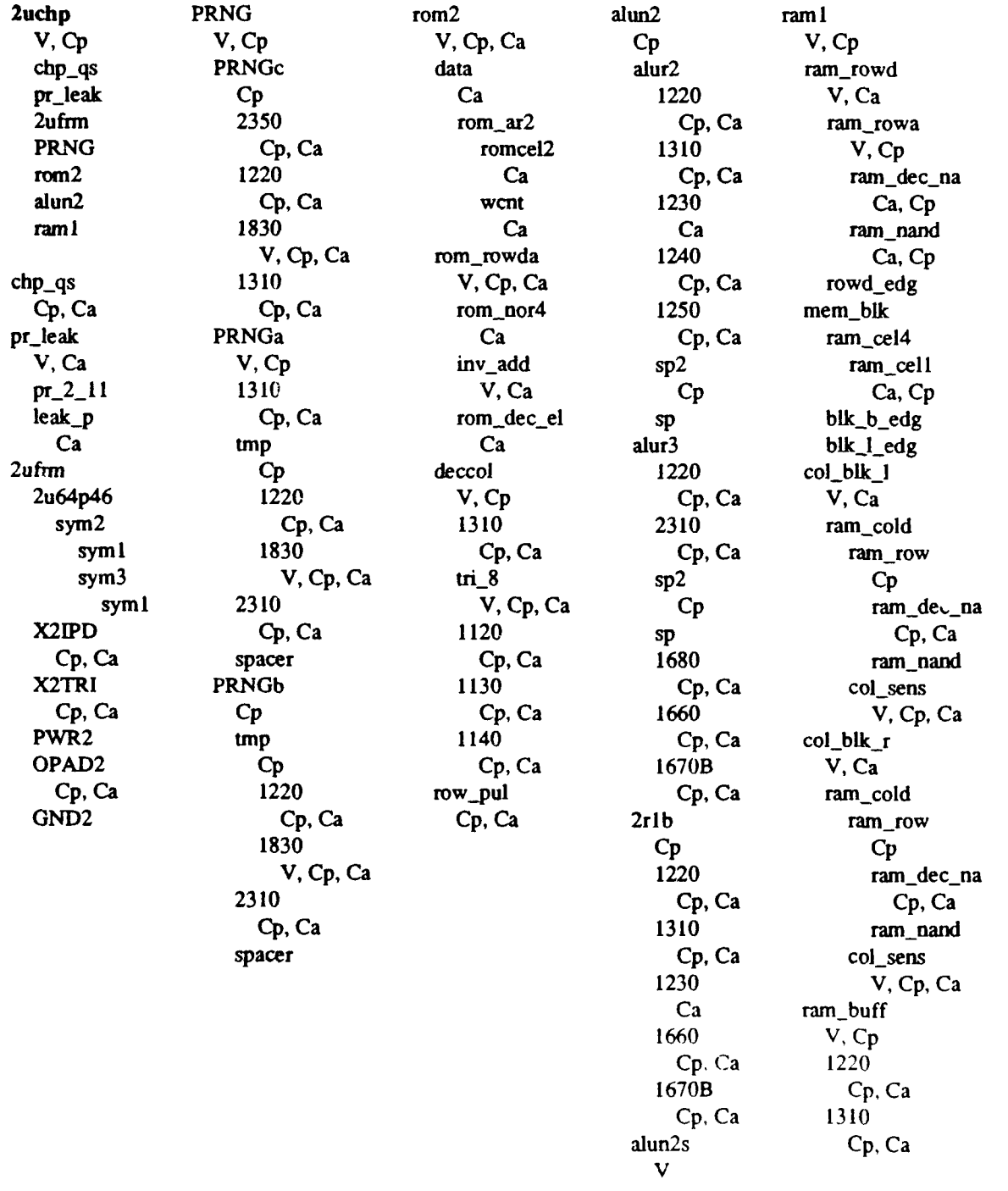



Figure 6.1 - The hierarchy of the 2uchp chip

The 2uchp chip contains the "hard" case that was mentioned in Chapter 5: the *2ufrm* cell is a padframe cell which overlaps the other major cells (*raml*, *rom2*, *alun2*, *PRNG*, *chp_qs*, and *pr_leak*). Another major cell, the *alun2* cell, has overlapping instances. In *alun2*, the instance of *alun2s* overlaps the other instances over most of their area. Therefore to extract the entire chip hierarchically, it is important to avoid flattening the areas of overlap.

### 6.1.2. The 2uchp test results

The cells of the chip were used to test the functionality of iCHARM. All of the cells were put through the flat extraction, parasitic extraction, power bus extraction, and hierarchical extraction modes of the extractor. Of most interest was the verification of the hierarchical extraction capability and the performance comparison of the hierarchical mode versus the flat mode. The test results for the larger cells of 2uchp are shown in Table 6.1.

Out of the 74 total cells, a number of leaf cells are simply connector or feed-through cells and do not contain any transistors, and a number of cells are simple logic gates (nand, nor, inv). To reduce the size of Table 6.1, none of these cells are reported. The second and third columns of Table 6.1 report the number of transistors and nets for each cell. Next, the *regularity* of a cell is defined as

$$\text{regularity} = \frac{\text{number of instances of the cell}}{\text{number of unique subcells}}$$

For instance, a cell may have one instance each of five subcells and four instances of another subcell which results in a regularity factor of 9/6 = 1.5. The number of overlap-rects of a cell is also reported in Table 6.1.

The performance results for flat and hierarchical extractions are reported in the form: input/pre-process/final. All reported times are the user times taken by the process after the given step has completed. The input time refers to the time taken to read the layout in. Some preprocessing is done before the actual extraction; the time after the preprocessing step is reported. In flat extraction, the preprocessing step is the flattening of the layout, and in hierarchical extraction, the overlap analysis is the preprocessing step. The final time is the total time taken by the process. In other words, all times are reported relative to the start time of zero, therefore, "0:04/0:12/2:13" means the process took 4 seconds of input time, then 8 seconds for preprocessing, followed by 2:01 minutes to do the extraction.

**Table 6.1 - iCHARM Test Results with 2uchp Cells**

| Cell name | Tran-sistors | Nets | Reg-ularity | Overlap-rects | Flat | | Hierarchical | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Time | Mem. | Time | Mem. |
| col_sens | 16 | 18 | 50.0 | 26 | 0:00/0:00/0:03 | 46.7 | 0:00/0:01/0:04 | 84.6 |
| ram_cell | 6 | 8 | 7.0 | 0 | 0:00/0:00/0:00 | 9.5 | 0:00/0:00/0:00 | 12.8 |
| ram_cel4 | 24 | 19 | 4.0 | 0 | 0:00/0:00/0:02 | 29.9 | 0:00/0:00/0:01 | 29.1 |
| ram_nand | 8 | 22 | 20.5 | 72 | 0:00/0:00/0:01 | 16.9 | 0:00/0:00/0:01 | 25.5 |
| ram_dec_na | 10 | 7 | 25.0 | 47 | 0:00/0:00/0:00 | 15.1 | 0:00/0:00/0:01 | 24.5 |
| ram_rowa | 28 | 39 | 2.0 | 57 | 0:00/0:00/0:02 | 46.4 | 0:00/0:01/0:03 | 69.6 |
| mem_blk | 3072 | 1122 | 58.7 | 1 | 0:00/0:07/4:47 | 3022.8 | 0:00/0:00/0:18 | 945.7 |
| ram_rowd | 448 | 145 | 18.5 | 18 | 0:01/0:02/0:39 | 502.0 | 0:01/0:01/0:11 | 393.3 |
| alun2s | 0 | 4 | 3.0 | 301 | 0:00/0:00/0:00 | 1.8 | 0:00/0:00/0:00 | 2.6 |
| tri_8 | 14 | 16 | 40.0 | 1 | 0:00/0:00/0:01 | 28.3 | 0:00/0:00/0:02 | 57.0 |
| rom_dec_el | 4 | 8 | 9.0 | 10 | 0:00/0:00/0:00 | 4.4 | 0:00/0:00/0:00 | 6.0 |
| inv_add | 6 | 4 | 11.5 | 16 | 0:00/0:00/0:00 | 7.9 | 0:00/0:00/0:00 | 14.1 |
| rom_nor4 | 24 | 26 | 70.0 | 51 | 0:00/0:00/0:02 | 27.1 | 0:00/0:00/0:01 | 39.6 |
| romcel2 | 2 | 7 | 2.0 | 12 | 0:00/0:00/0:00 | 2.4 | 0:00/0:00/0:00 | 3.6 |
| rom_ar2 | 256 | 305 | 80.0 | 370 | 0:00/0:00/0:13 | 178.9 | 0:00/0:00/0:03 | 207.8 |
| row_pul | 2 | 6 | 2.5 | 6 | 0:00/0:00/0:00 | 3.0 | 0:00/0:00/0:00 | 5.5 |
| rom_rowda | 110 | 55 | 17.7 | 49 | 0:01/0:01/0:07 | 106.3 | 0:01/0:01/0:05 | 134.9 |
| data | 256 | 49 | 128.5 | 75 | 0:01/0:01/0:19 | 226.6 | 0:01/0:03/0:15 | 459.5 |
| PRNGc | 47 | 28 | 4.6 | 4 | 0:01/0:01/0:06 | 76.7 | 0:01/0:01/0:06 | 72.2 |
| OPAD2 | 23 | 5 | 172.5 | 4 | 0:01/0:01/0:03 | 96.4 | 0:01/0:02/0:07 | 149.7 |
| X2TRI | 34 | 10 | 215.0 | 1 | 0:01/0:01/0:04 | 124.7 | 0:01/0:03/0:09 | 179.4 |
| X2IPD | 16 | 5 | 126.0 | 1 | 0:00/0:01/0:02 | 74.5 | 0:00/0:01/0:04 | 105.3 |
| chp_qs | 2 | 8 | 16.0 | 5 | 0:00/0:00/0:00 | 8.9 | 0:00/0:00/0:00 | 15.1 |
| 2ufrm | 922 | 176 | 9.0 | 1379 | 0:04/0:12/2:13 | 2429.7 | 0:04/0:09/3:32 | 6769.8 |
| ram_row | 18 | 27 | 1.3 | 58 | 0:00/0:00/0:02 | 33.0 | 0:00/0:00/0:02 | 54.3 |
| ram_cold | 50 | 53 | 1.5 | 69 | 0:01/0:01/0:10 | 108.1 | 0:01/0:01/0:07 | 140.1 |
| ram_buff | 12 | 9 | 3.5 | 54 | 0:00/0:00/0:02 | 35.4 | 0:00/0:00/0:01 | 35.9 |
| col_blk_r | 400 | 213 | 10.3 | 10 | 0:01/0:03/1:20 | 840.9 | 0:01/0:02/0:11 | 308.3 |
| col_blk_l | 400 | 213 | 10.3 | 9 | 0:01/0:03/1:20 | 840.8 | 0:01/0:02/0:11 | 307.9 |
| 2r1b | 40 | 28 | 4.5 | 10 | 0:01/0:01/0:06 | 71.4 | 0:00/0:01/0:05 | 73.3 |
| alur3 | 108 | 90 | 3.3 | 4 | 0:01/0:01/0:17 | 213.5 | 0:01/0:01/0:06 | 97.3 |
| alur2 | 110 | 116 | 3.9 | 3 | 0:01/0:01/0:14 | 185.5 | 0:01/0:01/0:05 | 95.2 |
| deccol | 122 | 63 | 17.7 | 29 | 0:01/0:02/0:21 | 244.3 | 0:01/0:02/0:08 | 161.6 |
| tmp | 28 | 21 | 2.3 | 33 | 0:00/0:00/0:03 | 38.4 | 0:00/0:01/0:03 | 50.5 |
| PRNGb | 261 | 156 | 6.0 | 29 | 0:01/0:01/0:29 | 372.7 | 0:01/0:01/0:06 | 91.9 |
| PRNGa | 278 | 163 | 6.3 | 27 | 0:01/0:01/0:32 | 410.1 | 0:01/0:02/0:07 | 112.0 |
| ram1 | 7404 | 2561 | 76.3 | 29 | ?? | O.M. | 0:04/0:08/0:56 | 2071.5 |
| alun2 | 378 | 205 | 28.0 | 100 | 0:02/0:04/0:50 | 603.4 | 0:02/0:04/0:57 | 2750.3 |
| rom2 | 504 | 112 | 25.9 | 66 | 0:04/0:06/0:58 | 593.7 | 0:04/0:10/0:45 | 903.5 |
| PRNG | 586 | 332 | 9.4 | 22 | 0:01/0:03/1:09 | 859.8 | 0:02/0:03/0:24 | 938.5 |
| 2uchp | 9796 | 3320 | 131.4 | 0 | ?? | O.M. | 0:21/1:36/13:18 | 14191.2 |

Input format: Oct format
Parasitics extracted: Capacitance only
Hardware: Vax 3500
Time (user) reported in units of minutes:seconds, see text for format
Memory reported in units of kbytes
O.M. = Out of Memory (the process aborted)

Since the memory allocation in iCHARM was handled manually, it was possible to monitor the amount of memory that was allocated for each run of the program. The maximum amount of memory that was allocated during the process is reported in Table 6.1.

### 6.1.3. Interpretation of the test results

In looking over the test results in Table 6.1, some interpretations and conclusions can be formed from the data.

First of all, the overhead in the preprocessing step to handle overlap is minimal. In most cases the time to process the overlap is less than the time to flatten the entire cell. A good example of this is the *2ufrm* cell, where it took 12 seconds to flatten but only 9 seconds to process the overlap. Most cells in Table 6.1 take almost negligible time to process the overlap: from "zero" (less than 1 second) to 2 seconds.

To compare the times for hierarchical extraction versus flat extraction, it makes sense to discuss only the cells that have runtimes greater than 10 seconds. Measurement errors for the runtimes less than 10 seconds are a large percentage of the total runtime, so the shorter results cannot be trusted for comparisons. Of the results in Table 6.1, there are 18 cells where the total flat extraction time took longer than 10 seconds. These cells can be classified into three groups.

The best results show the advantages of hierarchical analysis: the hierarchical extraction takes less time and less memory. This occurs for the largest number of cells, 11 cells in all: {*mem_blk*, *ram_rowd*, *col_blk_r*, *col_blk_l*, *alur3*, *alur2*, *deccol*, *PRNGb*, *PRNGa*, *raml*, *2uchp*}. The extraction has a few overlap-rects, which means there is little or no overlap for the cell. In fact, the *mem_blk* cell has virtually no overlap; therefore, the speed-up is almost 16-fold, and one-third of the memory is used. For the *raml* and *2uchp* cells, hierarchical extraction produces an extracted result where the flat analysis failed to produce any result.

When overlap is present, the results are less impressive; five cells still have faster hierarchical extraction times but more memory is actually used: {*rom_ar2*, *data*, *ram_cold*, *rom2*, *PRNG*}. All of the cells end up having significant lists of overlap-rects. More memory must be used to calculate and create the overlap-rects. The extraction process itself is slowed since during extraction each rect for the cell must be checked to see if it touches an overlap-rect. Nevertheless, the savings due to hierarchy are great enough so that the runtime is still faster.

Finally there are two cells for which hierarchical analysis is worse in terms of both speed and memory usage: {*2ufrm*, *alun2*}. As mentioned before the padframe is the "hard" test-case

for hierarchical extractors, and the *alun2* cell contains a significant amount of overlap. However, it does appear that for the 18 test-cases, most of the cells benefit from being extracted hierarchically with the methods employed by iCHARM.

## 6.2. Future Extensions

A number of extensions to the existing implementation of iCHARM as presented in this thesis are possible.

### 6.2.1. Output results in Oct

At present, iCHARM reports the extracted results in the form of a Spice-format file. As discussed in the section on the Oct front-end, there is a need to write the extracted results back into the Oct database. Such a system would have all of the advantages of a fully integrated system. To implement such a feature, a "simulator" facet would be created that contains the extracted circuit connectivity information; the target simulator would read this facet. To be successful, the extractor and the target simulator would have to agree on a suitable *simulator policy* so that the extracted information is complete, and the simulator would know where to expect certain data. The Spice policy as shown in the Oct Tools manual [12] could be used as the first cut toward developing a customized simulator policy.

### 6.2.2. Coupling capacitance

The parasitics computed for iCHARM are limited to the substrate capacitance of each net and the interconnect resistance. The coupling capacitance between conductors of two nets becomes more significant as the minimum feature size on a chip is made smaller. This causes the sidewall or fringing-field capacitance components of conductors to become larger and thus increases the coupling capacitance between conductors.

The major problem with computing the coupling capacitances of a circuit is that the process is computationally expensive. Since every rectangle of a net must be checked against every other rectangle of the surrounding nets, a thorough and complete analysis is easily an $O(N^2)$ process. In order to reduce the amount of computation and since coupling capacitance is inversely proportional to the distance between two conductors, the coupling capacitance between rectangles that are beyond a reasonable proximity of one another is not calculated. However, scanline algorithms are not very well-suited for the "proximity analysis" needed to compute

coupling capacitances. Other extraction algorithms — 4-D trees for example — are better suited for the problem of finding all rectangles within a set separation distance from a given rectangle.

### 6.2.3. Multiprocessor implementation

There are two ways to speed up the circuit extraction process. The first has been exploited in iCHARM: using hierarchy to eliminate the analysis of repeated cells. A second speed-up can be gained by exploiting the time-parallelism of the extraction process, namely, to implement the hierarchical extractor on a parallel machine. The goal in such an implementation would be to schedule the extraction of one hierarchical cell using an available processor such that the computational load between processors would be balanced. Obviously, the extraction of the largest cell of a design could take longer than all of the other cells combined. In addition to distributing the extraction of many cells among processors, a capability to split the extraction of a single large cell between processors would also be necessary.

### 6.2.4. Power bus extraction and modeling

The capacitance and resistance extraction and modeling done by the JET program for reliability analysis and the resistance extraction mode of iCHARM basically accomplish the same task. The obvious plan would be to combine the two programs. The JET program is customized to analyze only power busses but is not as general in its approach as iCHARM. The JET program splits up the rectangles of the power bus into a restricted set of primitive shapes that it can model. It cannot analyze certain configurations if they cannot be decomposed into one of its primitives. Therefore, iCHARM is more general, but not as accurate as JET.

To combine the two programs, iCHARM could do its rectangle decomposition step and resistance extraction for the areas of a power bus that do not correspond to a JET primitive region. This would add some flexibility to the modeling system.

## 6.3. Conclusions

This thesis began with the introduction of hierarchical design methods as a way to increase productivity and reduce the complexity of the design task. Similarly, hierarchical design tools reduce the unit of analysis from the entire chip to one cell at a time.

Circuit extraction is defined as the transformation of layout into circuit information. It is the link between the design of ICs and their verification through simulation. In addition to verifying the functionality of a layout, it is important to accurately model any unexpected parasitic

effects. The iCHARM program was developed to extract CMOS circuits with parasitics, using hierarchy to reduce the time and memory required for the extraction.

To efficiently represent the input layout, iCHARM builds hierarchical data structures consisting of cell and instance structures. The extractor reads the input layout in two formats: the CIF format and Oct database format. The CIF format organizes the description of the layout in cells; transformation matrices are used to describe how a cell is instantiated. The CIF format is meant to be used as an interchange format to describe a finished layout; it is not meant to be used as a design language or database. Cells described in the Oct format, which was developed at UC Berkeley, can be part of an integrated design system built around a common database. Files in the Oct format can be created by a program such as Vem, the standard Oct-tools graphical editor. Application programs such as iCHARM can read Oct-format files by calling Oct access routines.

The geometric extraction step, in which the transistors and electrical nets of the circuit are identified, is the basis of the extraction process. The data structure used to represent the layout implies the algorithm that must be used for geometric extraction which also implies the performance and memory usage of the step. Three data structures were surveyed: 4-D trees, scanlines, and corner-stitching. The 4-D trees are the most suitable for fast geometric queries typical of interactive applications. There is some overhead in the technique to keep the tree balanced for optimum performance. Scanlines are a simple technique since the underlying data structure is a simple linked list of rectangles. Since all of the rectangles of the layout are analyzed in left-to-right order in one big procedure, it is most suitable for batch processing of rectangles. Finally, the corner-stitched data structuring technique optimizes the nearest neighbor query, but has difficulty with a large number of layers.

The iCHARM program used an existing program as its basis which uses a scanline algorithm for geometric extraction. The basic scanline technique is implemented by processing the intersecting rectangles in the AddToScanline() routine, and the DeleteFromScanline() routine is used to do any final processing of each rectangle. Applications of the basic scanline algorithm are shown for net and transistor extractions.

The parasitic extraction mode in iCHARM extracts the substrate capacitance of each net and the resistance through each net. This results in a distributed RC model of each interconnection net. A scanline technique is again used to create an electrical connectivity graph for the rectangles of each net. The graph links each rectangle with its adjacent rectangles. The contribution of each rectangle to the net's capacitance is calculated from the area and perimeter of the

rectangle. By identifying the knots and branches of the net, the resistance is calculated using the resistance formulas along each branch.

The power bus extraction mode of iCHARM is used, along with the JET and CREST programs, to predict the reliability of a given chip. The mode extracts the power bus conductors and splits and renumbers each power bus contact point. A scanline algorithm is employed to collect "diffusion-islands" at each contact point.

Hierarchical extraction offers a savings in extraction time as well as memory usage. The overlap of instances is the principal problem with the technique since it destroys the independence of a previously extracted cell with its instantiation. Existing techniques for hierarchical extraction either convert an overlapped design into an equivalent nonoverlapped one where simpler techniques can be used for extraction, or the overlapping is dealt with directly. In this case the "padframe" case causes the entire hierarchy to be flattened.

Terminal structs are used by iCHARM to extend the flat extraction procedure into a hierarchical one. Terminals allow cells to connect to nets outside of the cell instance. In order to handle overlap, a preprocessing step is done to find all the unique situations in which a cell's instances are overlapped. This creates overlap-rects for each cell. During extraction, external terminals are created for any rect of the cell that touches an overlap-rect.

Capacitance adjustments for abutting and overlapping connections to internal terminals must be done for hierarchical capacitance extraction. To implement hierarchical power bus extraction, a preprocessing step is added to mark all the nets throughout the hierarchy that connect to the power and ground nets of the top-level cell.

A full-chip layout was used to test the program. The overhead of the hierarchical preprocessing step was minimal. The hierarchical extraction results showed that most cells have little overlap, and most cells are extracted in less time and with less memory with hierarchical methods.

# APPENDIX

# iCHARM USER MANUAL

This section consists of a user manual for the iCHARM Hierarchical CMOS Extraction program. The capabilities of the program are given in Chapter 1 of this thesis. The first section of the manual describes how to run the program and its command line options. The next section describes the iCHARM-specific requirements for the CIF or Oct format input files. The iCHARM program reads a technology file which describes the process-dependent parameters of each layer. The format of the technology file is given in the next section, as well as some notes about which layer names are required by the program.

## A.1. Running iCHARM

The usage string for the program is

**iCHARM -i (<ciffile> | <octcell:view>) [-h] [-r] [-t <techfile>]**
**[-p [<pwrbusfile>]] [-v pwrname] [-g gndname]**

The Spice format output file is written on stdout. Any warning or error messages are printed on stderr.

Only the -i flag and its argument must be specified. All other options are optional and default values will be assigned if a given option is not specified. Each option flag has the following meaning:

**-i (<ciffile> | <octcell:view>)**

The input CIF file or Oct facet is specified with this option. This information must be specified or the program will exit.

An Oct facet is assumed if the string following the -i flag has a colon (:) anywhere in it. In this case, the facet in the file "<octcell>/<view>/contents;" will be read. Both the cell and view name must be specified; the contents facet is assumed. The facet specified will be used as the top-level cell extracted.

A CIF format file is assumed if the <ciffile> string is specified without any colon character. The final command before the End command in the CIF file must be a single call to the top-level cell in the file. For example, if the cell (or symbol in CIF terminology) 57 is the top-level cell in the hierarchy of a given CIF file, then the end of the file should resemble

```
...
DS 57;
...
DF;
C 57;
E
```

**-h** If this flag is specified, the extraction is done hierarchically: the output Spice file will have .SUBCKT blocks for each hierarchical cell. By default (if the flag is not specified), the extraction is done flat, with any hierarchy in the input flattened into a single level of hierarchy that contains all of the rectangles of the layout.

**-r** This flag controls the extraction of the parasitics of the interconnection nets. The substrate capacitance of each net is always extracted. If the -r flag is specified, however, the resistance through each net is also calculated and reported in the output file. This requires more calculation, so the use of the -r option slows the performance of the extractor.

**-t <techfile>**

iCHARM assumes the existence of a technology file in the current directory. Since the default name of the file is "tech," the program looks for such a file in the current directory. The -t option changes the name of the technology file that the program uses to its argument, "<techfile>."

**-p [<pwrbusfile>]**

iCHARM has a special option to extract the rectangles that make up the power bus nets (Power and Ground). The power bus processing is done only when the -p option is specified. If only the -p flag is specified, a CIF format file of the extracted power bus rectangles is output to the default filename of "pwrbus.cif." If an argument is given along with the -p flag, then the power bus file is named "<pwrbusfile>."

**-v <pwrname> -g <gndname>**

The two power and ground nets are recognized by their names: the default name for the

power net is "Vdd" and the default name for the ground net is "GND." The -v and -g options allow the default names to be changed for the power net name and the ground net name, respectively. A net with the power (ground) net name is given the reserved net number of "1" ("0," respectively) in the output Spice file. The power bus extraction module that was mentioned above identifies the two power bus nets that are extracted by the same naming and recognition mechanism.

## A.2. Input Format

iCHARM accepts input files in either the CIF format or the Oct format. In so doing, there are some iCHARM-specific extensions, options, and caveats that must be described.

### A.2.1. CIF format

In CIF, user extension commands begin with the digit "9" that allow "tailor-made" options specific to a program to be defined. The following user extension commands are recognized by iCHARM:

**9 <cellname>;**

When this command occurs between a "DS" and a "DF" command, the cell being defined is given the name of <cellname>. If hierarchical extraction is done, <cellname> is used to name the cell's .SUBCKT in the output Spice file.

**94 <label> <x> <y> <layername>;**

The "94" command is used to define a *label* at the point <x>,<y> on the layer <layername>. Labels are used to name nets. When the extraction is done, the net that has a rectangle at the point <x>,<y> on the layer <layername> is given the name of <label>. At the very minimum, the power bus nets should be named so that the power net recognition mechanism described above will work.

Labels may be defined for all cells, but only labels in the top-level cell are actually used to name nets. All other labels are parsed in the input file, but are ignored. This is done to reduce the number of named nets and simplify the output file.

### A.2.2. Oct format

Oct formal terminals correspond to the "94" commands in CIF files; that is, formal terminals in a facet are used to create iCHARM labels. In addition, Oct labels (the "label" octObject) are also used to create iCHARM labels.

### A.2.3. Defining a new technology in Oct

The input Oct facet must have layer names that are accepted by iCHARM. The accepted layer names are given in the next section. To do this, a new technology type has to be defined. The Oct tools manual [12] documents the procedure to define a new technology, but the important points will be covered here.

To create a new technology, the primary thing that has to be done is to create a facet named "tap.views:<view>:contents." Using a pattern file that defines each layer, the tap.views facet can be created by the Oct-tools program "pat2tap". This facet has to be placed under a directory with the name of the technology. For instance, to define a new technology with the name "mosis" for physical (layout) editing, one would create the facet "tap.views:physical:contents" under the directory "mosis."

The tap.views facet is used by Vem to determine how to display the various layers. The display characteristics are determined by the attachments of the tap.views facet as shown in Figure A.1. To display the layers in Vem (on a color display), Vem looks for a "GENERIC-COLOR" bag attached to the "DISPLAY-LOOKS" bag that is attached to each layer in the tap.views facet. To obtain a print-out of a facet, the "oct2ps" program may be used to produce a Postscript file that displays the facet. To do so, oct2ps looks for a "Postscript-BW" bag in the "DISPLAY-LOOKS" bag in the tap.views facet. A final note: in the Octtools 3.0 release, a layer named "PLACE" must be defined in the tap.views facet or various tools will not work.

The entire directory for the new technology has to be under a directory that is called the "default technology directory." This directory can contain the definitions for all of the technologies that are defined by the user. The -T option in many Oct programs takes the default technology directory as its argument. For instance, the -T is used in the vulcan program command line:

"vulcan -T ~/octvem/technology mycell:physical"

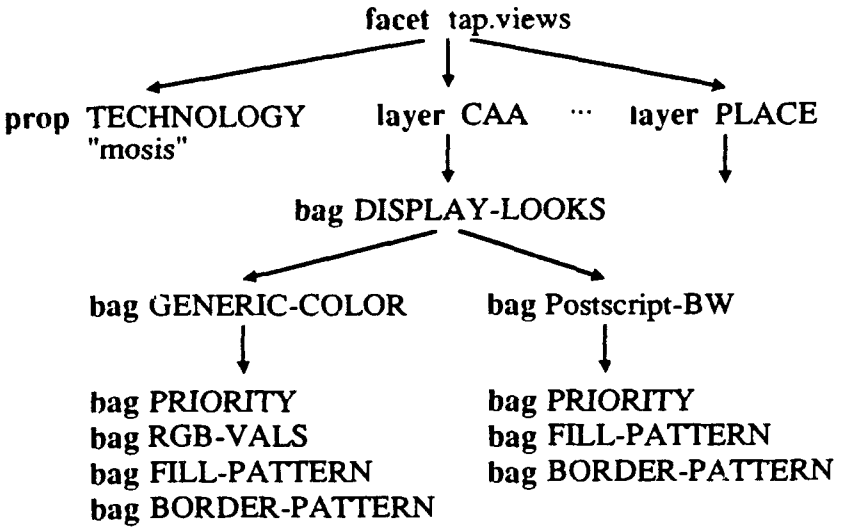



**Figure A.1 - The attachments for the facet tap.views for display with Vem and Oct2ps**

The user's .Xdefaults file has to be modified when a new technology is used. The "vem.technology" variable has to be set to the default technology directory mentioned above, and the "vem.deftechname" should be set to the name of the new technology.

### A.3. Technology File

The technology file in iCHARM defines the process parameters for each layer. The parser that reads the file is crude so the file is kept simple, and no comments are accepted in the file. An example of a valid technology file is given in Figure A.2.

The first section of the technology file determines which layers connect with other layers. The first line has the total number of layers for the process. One line for each layer follows that gives information about each layer. The first column is the number of other layers that the layer-for-this-line connects with. The second column should be "1" if the layer is a contact layer. The rest of the columns have the layer numbers of the layers that this layer connects with.

The next section in the technology file describes the process parameters of each layer that are used in calculating the extracted parasitic values. Again there is one line for each layer. The first column is the sheet resistance for the layer, in the units of ohms per square. The second

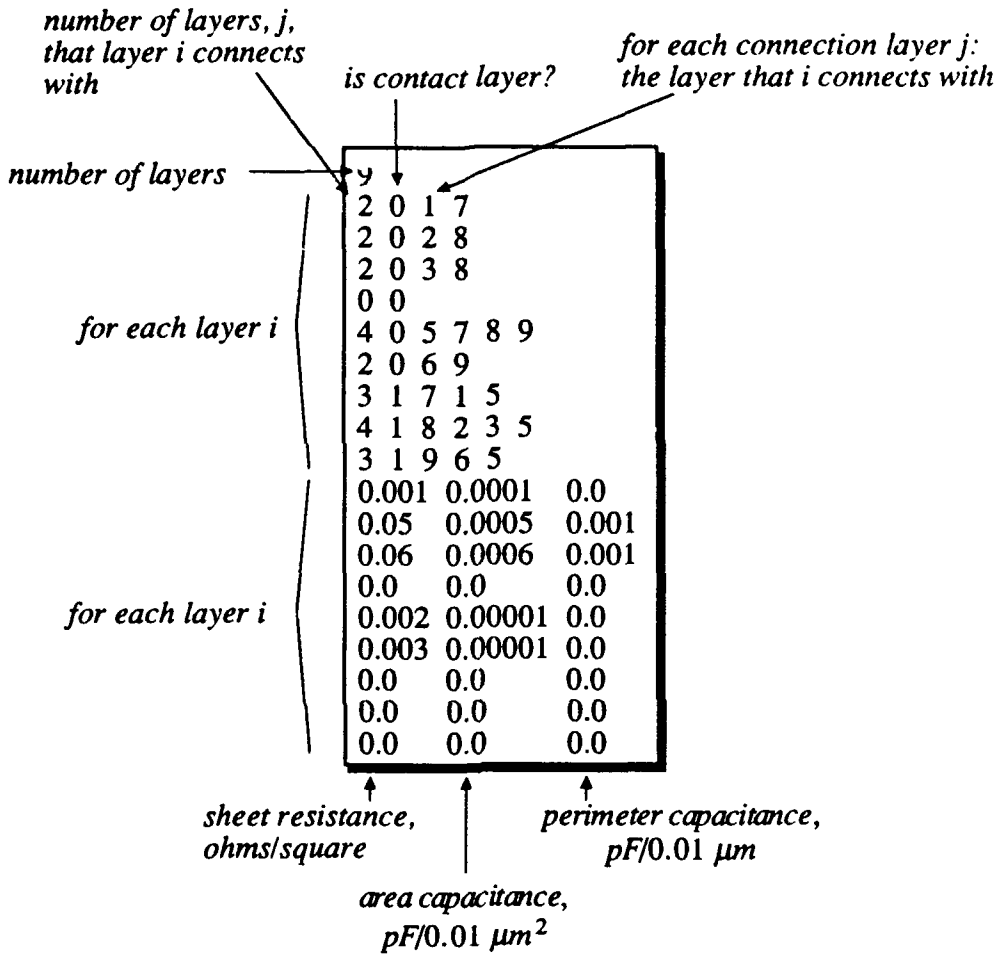



**Figure A.2 - A sample technology file**

column is the capacitance per area for the layer, in pico-farads per 0.01 micron$^2$. The third column holds the value for the perimeter capacitance (capacitance per unit length) of the layer, in pico-farads per 0.01 micron. The values shown in Figure A.2 are for a generic CMOS process based on typical values from Weste [17].

## A.4. Layer Names

The layer names that are recognized by iCHARM are, unfortunately, hard-coded into the program. The names and their meaning are given in Table A.1. The iCHARM program uses a subset of the "standard" MOSIS CMOS layer names. All other layer names not found in Table A.1 are ignored by iCHARM.

**Table A.1 - iCHARM Layer Names**

| Layer name | Meaning |
|---|---|
| CAA | Active area (diffusion) |
| CPW | P-well |
| CPG | Polysilicon |
| CMF | First-level metal |
| CMS | Second-level metal |
| CCA | Contact between CAA and CMF |
| CCP | Contact between CPG and CMF |
| CVA | Contact between CMF and CMS (via) |

The CAA layer is used for both p-fets and n-fets. A P-well process is assumed here: N-diffusion is formed when a P-well is over a section of active area (CAA) and P-diffusion is assumed for active area rectangles without P-well areas.

# REFERENCES

[1] K.P. Belkhale and P. Banerjee, "PACE: A parallel VLSI circuit extractor on the Intel hypercube multiprocessor," *Proceedings of the International Conference on Computer-Aided Design*, pp. 326-329, 1988.

[2] K.P. Belkhale and P. Banerjee, "PACE2: An improved parallel VLSI extractor with parameter extraction," *Proceedings of the International Conference on Computer-Aided Design*, pp. 526-529, 1989.

[3] H. Cha, "Current density calculation using rectilinear region splitting algorithm for VLSI metal migration analysis," M.S. thesis, University of Illinois at Urbana-Champaign, Department of Electrical Engineering, in preparation.

[4] A. Gupta, "ACE: A circuit extractor," *Proceedings of the 20th Design Automation Conference*, pp. 721-725, 1983.

[5] P.M. Hall, "Resistance calculations for thin film patterns," *Thin Solid Films*, vol. 1, pp. 277-295, Mar. 1968.

[6] M. Horowitz and R.W. Dutton, "Resistance extraction from mask layout data," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-2, no. 3, pp. 145-150, July 1983.

[7] L. Kohn and S.W. Fu, "A 1,000,000 transistor microprocessor," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 54-55, Feb. 1989.

[8] D. Marple, M. Smulders and H. Hegen, "Tailor: A layout system based on trapezoidal corner stitching," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 1, pp. 66-90, Jan. 1990.

[9] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA:Addison-Wesley, 1980.

[10] F. Najm, R. Burch, P. Yang and I. Hajj, "CREST — a current estimator for CMOS circuits," *Proceedings of the International Conference on Computer-Aided Design*, pp. 204-207, 1988.

[11] M.E. Newell and D.T. Fitzpatrick, "Exploitation of hierarchy in analyses of integrated circuit artwork," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-1, no. 4, pp. 192-200, Oct. 1982.

[12] Oct Tools Distribution 3.0, Electronics Research Laboratory, University of California, Berkeley, 1989.

[13] J.B. Rosenberg, "Geographical data structures compared: A study of data structures supporting region queries," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-4, no. 1, pp. 53-67, Oct. 1985.

[14] W.S. Scott and J.K. Ousterhout, "Magic's circuit extractor," *Proceedings of the 22nd Design Automation Conference*, pp. 286-292, 1985.

[15] S.L. Su, "Extraction of MOS VLSI circuit models including critical interconnect parasitics," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Department of Electrical Engineering, 1987.

[16] T.G. Szymanski and C.J. Van Wyk, "Goalie: A space efficient system for VLSI artwork analysis," *IEEE Design and Test of Computers*, vol. 2, no. 3, pp. 64-72, June 1985.

[17] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*. Reading, MA:Addison-Wesley, 1985.